Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

Semantics and Pragmatics of Real-Time Maude

Peter Csaba Ölveczky · **José Meseguer**

Received: date / Accepted: date

Abstract At present, designers of real-time systems face a dilemma between expressiveness and automatic verification: if they can specify some aspects of their system in some automaton-based formalism, then automatic verification is possible; but more complex system components may be hard or impossible to express in such decidable formalisms. These more complex components may still be simulated; but there is then little support for their formal analysis. The main goal of Real-Time Maude is to provide a way out of this dilemma, while complementing both decision procedures and simulation tools. Real-Time Maude emphasizes ease and generality of specification, including support for distributed real-time object-based systems. Because of its generality, falling outside of decidable system classes, the formal analyses supported—including symbolic simulation, breadth-first search for failures of safety properties, and model checking of time-bounded temporal logic properties—are in general incomplete (although they are complete for discrete time). These analysis techniques have been shown useful in finding subtle bugs of complex systems, clearly outside the scope of current decision procedures. This paper describes both the semantics of Real-Time Maude specifications, and of the formal analyses supported by the tool. It also explains the tool's pragmatics, both in the use of its features, and in its application to concrete examples.

Keywords Rewriting logic · real-time systems · object-oriented specification · formal analysis · simulation · model checking

Peter Csaba Ölveczky
Department of Informatics
University of Oslo
Tel.: +47-22852498
Fax: +47-22852401
E-mail: peterol@ifi.uio.no

José Meseguer
Department of Computer Science
University of Illinois at Urbana-Champaign
Tel.: +1-217-3336733
Fax: +1-217-3339386
E-mail: meseguer@cs.uiuc.edu

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2007	**N/A**	**-**

4. TITLE AND SUBTITLE
Semantics and Pragmatics of Real-Time Maude

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Illinois at Urbana-Champaign Department of Computer Science 201 N. Goodwin Avenue Urbana IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
At present, designers of real-time systems face a dilemma between expressiveness and automatic verification: if they can specify some aspects of their system in some automaton-based formalism, then automatic verification is possible; but more complex system components may be hard or impossible to express in such decidable formalisms. These more complex components may still be simulated; but there is then little support for their formal analysis. The main goal of Real-Time Maude is to provide a way out of this dilemma, while complementing both decision procedures and simulation tools. Real-Time Maude emphasizes ease and generality of specification, including support for distributed real-time object-based systems. Because of its generality, falling outside of decidable system classes, the formal analyses supportedincluding symbolic simulation, breadth-first search for failures of safety properties, and model checking of time-bounded temporal logic properties are in general incomplete (although they are complete for discrete time). These analysis techniques have been shown useful in finding subtle bugs of complex systems, clearly outside the scope of current decision procedures. This paper describes both the semantics of Real-Time Maude specifications, and of the formal analyses supported by the tool. It also explains the tools pragmatics, both in the use of its features, and in its application to concrete examples.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**SAR**	**35**	

1 Introduction

At present, designers of real-time systems face a dilemma between expressiveness and automatic verification. If they can specify some aspects of their system in a more restricted automaton-based formalism, then automatic verification of system properties may be obtained by specialized model checking decision procedures. But this may be difficult or impossible for more complex system components which may be hard or impossible to express in such decidable formalisms. In that case, simulation offers greater modeling flexibility, but is typically quite weak in the kinds of formal analyses that can be performed. The main goal of Real-Time Maude is to provide a way out of this dilemma, while complementing both decision procedures and simulation tools.

On the one hand, Real-Time Maude can be seen as complementing tools based on timed and linear hybrid automata, such as UPPAAL [19,5], HyTech [15], and Kronos [32]. While the restrictive specification formalism of these tools ensures that interesting properties are decidable, such finite-control automata do not support well the specification of larger systems with different communication models and advanced object-oriented features. By contrast, Real-Time Maude emphasizes ease and generality of specification, including support for distributed real-time object-based systems. The price to pay for increased expressiveness is that many system properties may no longer be decidable. However, this does not diminish either the need for analyzing such systems, or the possibility of using decision procedures when applicable. On the other hand, Real-Time Maude can also be seen as complementing traditional testbeds and simulation tools by providing a wide range of formal analysis techniques and a more abstract specification formalism in which different forms of communication can be easily modeled and can be both simulated and formally analyzed. Finally, some tools geared toward modeling and analyzing larger real-time systems, such as, e.g., IF [6], extend timed automaton techniques with explicit UML-inspired constructions for modeling objects, communication, and some notion of data types. Real-Time Maude complements such tools not only by the full generality of the specification language and the range of analysis techniques, but, most importantly, by its simplicity and clarity: A simple and intuitive formalism is used to specify both the data types (by *equations*) and dynamic and real-time behavior of the system (by *rewrite rules*). Furthermore, the operational semantics of a Real-Time Maude specification is clear and easy to understand.

A key goal of this work is to document the tool's theoretical foundations, based on a simplified semantics of real-time rewrite theories [23,28] made possible by some recent developments in the foundations of rewriting logic [7]; these simplified theoretical foundations are explained in Section 3. We also give a precise description of the semantics of Real-Time Maude specifications and of its symbolic execution and formal analysis commands. Such semantics is given by means of a family of *theory transformations*, that associate to a real-time rewrite theory and a command a corresponding ordinary rewrite theory (a Maude [9, 10] system module) and a Maude command with the intended semantics (Section 5). Besides thus giving a precise account of the tool's *semantics*, we also explain and illustrate its *pragmatics* in several ways:

1. We discuss different *time domains* (both discrete and continuous) provided by the system, which also allows the user to define new such time domains in Maude modules.
2. We then explain the general methods by which *tick rules* for advancing time in the system can be defined.

3. We also explain some general techniques to specify *object-oriented* real-time systems in Real-Time Maude; such techniques have been developed through a good number of substantial case studies and have proved very useful in practice.
4. We give an overview of the tool's language features, commands, and analysis capabilities (Section 4).
5. We illustrate the tool's use in practice by means of two examples (Section 6).

Real-Time Maude specifications are *executable* formal specifications. Our tool offers various simulation, search, and model checking techniques which can uncover subtle mistakes in a specification. Timed *rewriting* can simulate *one* of the many possible concurrent behaviors of the system. Timed *search* and *time-bounded linear temporal logic model checking* can analyze *all* behaviors—relative to a given *time sampling strategy* for dense time as explained in Section 4.2.1—from a given initial state up to a certain time bound. By restricting search and model checking to behaviors up to a certain time bound and with a given time sampling strategy, the set of reachable states is typically restricted to a finite set, which can be subjected to model checking. Search and model checking are "incomplete" for dense time, since there is no guarantee that the chosen time sampling strategy covers all interesting behaviors. However, all the large systems we have modeled in Real-Time Maude so far have had a discrete time domain, and in this case search and model checking can completely cover all behaviors from the initial state. For further analysis, the user can write his/her own specific analysis and verification strategies using Real-Time Maude's reflective capabilities.

The Real-Time Maude tool described in this paper is a mature and quite efficient tool available free of charge (with sources, a tool manual, examples, case studies, and papers) from `http://www.ifi.uio.no/RealTimeMaude`. The tool has been used in a number of substantial applications, a subset of which is listed in Section 6.4. Real-Time Maude is based on earlier theoretical work on the rewriting logic specification of real-time and hybrid systems [23,28], and has benefited from the extensive experience gained with an earlier tool prototype [27,23], which was applied to specify and analyze a sophisticated multicast protocol suite [23,26]. As mentioned above, the current tool has simpler foundations based on more recent theoretical advances. Furthermore, thanks to the efficient support of breadth-first search and of on-the-fly LTL model checking in the underlying Maude 2 system [10], on top of which it is implemented, the current tool supports symbolic simulation, search for violations of safety properties, and model checking of time-bounded temporal logic properties with good efficiency.

2 Equational Logic, Rewriting Logic, and Maude

Since Real-Time Maude extends Maude and its underlying rewriting logic formalism, we first present some background on equational logic, rewriting logic, and Maude.

2.1 Equational and Rewriting Logic

Membership equational logic (**MEL**) [22] is a typed equational logic in which data are first classified by *kinds* and then further classified by *sorts*, with each kind k having an associated set S_k of *sorts*, so that a datum having a kind but not a sort is understood as an *error* or *undefined* element. Given a **MEL** signature Σ, we write $\mathbb{T}_{\Sigma,k}$ and $\mathbb{T}_{\Sigma}(X)_k$ to denote, respectively, the set of ground Σ-terms of kind k, and of Σ-terms of kind k over

variables in X, where $X=\{x_1:k_1,\dots,x_n:k_n\}$ is a set of kinded variables. *Atomic formulas* have either the form $t=t'$ (Σ-equation) or $t:s$ (Σ-membership) with $t,t'\in\mathbb{T}_\Sigma(X)_k$ and $s\in S_k$; and Σ-*sentences* are universally quantified Horn clauses on such atomic formulas. A $\mathbf{MEL}$ *theory* is then a pair (Σ,E) with E a set of Σ-sentences. Each such theory has an initial algebra $\mathbb{T}_{\Sigma/E}$ whose elements are equivalence classes of ground terms modulo provable equality.

In the general version of rewrite theories over $\mathbf{MEL}$ theories defined in [7], a *rewrite theory* is a tuple $\mathscr{R}=(\Sigma,E,\varphi,R)$ consisting of: (i) a $\mathbf{MEL}$ theory (Σ,E); (ii) a function $\varphi\colon\Sigma\to\mathscr{P}_{\mathrm{f}}(\mathbb{N})$ assigning to each function symbol $f\colon k_1\cdots k_n\to k$ in Σ a set $\varphi(f)\subseteq\{1,\dots,n\}$ of *frozen argument positions*; (iii) a set R of (universally quantified) labeled conditional rewrite rules r having the general form

$$(\forall X)\ r:\ t\longrightarrow t'\ \textbf{if}\ \textstyle\bigwedge_{i\in I}p_i=q_i\ \wedge\ \bigwedge_{j\in J}w_j:s_j\ \wedge\ \bigwedge_{l\in L}t_l\longrightarrow t'_l$$

where, for appropriate kinds k and k_l, $t,t'\in\mathbb{T}_\Sigma(X)_k$ and $t_l,t'_l\in\mathbb{T}_\Sigma(X)_{k_l}$ for $l\in L$.

The function φ specifies which arguments of a function symbol f *cannot be rewritten*, which are called *frozen positions*. Given a rewrite theory $\mathscr{R}=(\Sigma,E,\varphi,R)$, a *sequent* of $\mathscr{R}$ is a pair of (universally quantified) terms of the same kind t,t', denoted $(\forall X)\,t\longrightarrow t'$ with $X=\{x_1:k_1,\dots,x_n:k_n\}$ a set of kinded variables and $t,t'\in\mathbb{T}_\Sigma(X)_k$ for some k. We say that $\mathscr{R}$ *entails* the sequent $(\forall X)\,t\longrightarrow t'$, and write $\mathscr{R}\vdash(\forall X)\ t\longrightarrow t'$, if the sequent $(\forall X)\ t\longrightarrow t'$ can be obtained by means of the inference rules of reflexivity, transitivity, congruence, and nested replacement given in [7].

To any rewrite theory $\mathscr{R}=(\Sigma,E,\varphi,R)$ we can associate a Kripke structure $\mathscr{K}(\mathscr{R},k)_{L_\Pi}$ in a natural way provided we: (i) specify a kind k in Σ so that the set of *states* is defined as $\mathbb{T}_{\Sigma/E,k}$, and (ii) define a set Π of (possibly parametric) *atomic propositions* on those states; such propositions can be defined equationally in a protecting extension $(\Sigma\cup\Pi,E\cup D)\supseteq(\Sigma,E)$, and give rise to a *labeling function* L_Π on the set of states $\mathbb{T}_{\Sigma/E,k}$ in the obvious way. The *transition relation* of $\mathscr{K}(\mathscr{R},k)_{L_\Pi}$ is the one-step rewriting relation of $\mathscr{R}$, to which a self-loop is added for each deadlocked state. The semantics of linear-time temporal logic (LTL) formulas is defined for Kripke structures in the well-known way (e.g., [8,10]). In particular, for any LTL formula ψ on the atomic propositions Π and an initial state $[t]$, we have a satisfaction relation $\mathscr{K}(\mathscr{R},k)_{L_\Pi},[t]\models\psi$ which can be model checked, provided the number of states reachable from $[t]$ is finite. Maude [10] provides an explicit-state LTL model checker precisely for this purpose.

2.2 Maude and its Formal Analysis Features

A Maude module specifies a rewrite theory $(\Sigma,E\cup A,\varphi,R)$, with E a set of conditional equations and memberships, and A a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed *modulo* the axioms A. Intuitively, the theory $(\Sigma,E\cup A)$ specifies the system's state space as an algebraic data type, and each rewrite rule r in R specifies a (family of) *one-step transition(s)* from a substitution instance of t to the corresponding substitution instance of t', *provided* that the substitution satisfies the condition of the rule. The rewrite rules are applied *modulo* the equations $E\cup A$.[1]

[1] Operationally, a term is reduced to its E-normal form modulo A before any rewrite rule is applied in Maude. Under the coherence assumption [31] this is a complete strategy to achieve the effect of rewriting in $E\cup A$-equivalence classes.

We briefly summarize the syntax of Maude. *Functional* modules and *system* modules are, respectively, **MEL** theories and rewrite theories, and are declared with respective syntax `fmod ... endfm` and `mod ... endm`. *Object-oriented* modules provide special syntax to specify concurrent object-oriented systems, but are entirely reducible to system modules; they are declared with the syntax `(omod ... endom)`.[2] Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported submodules can be added. One can also declare *sorts*[3], *subsorts*, and *operators*. Operators are introduced with the `op` keyword. They can have user-definable syntax, with underbars '`_`' marking the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. The operator attribute `ctor` declares that the operator is a *constructor*, as opposed to a *defined function*. This attribute does not have any computational effect in Real-Time Maude. There are three kinds of logical statements: *equations*, introduced with the keywords `eq` and, for conditional equations, `ceq`; *memberships*, declaring that a term has a certain sort and introduced with the keywords `mb` and `cmb`; and *rewrite rules*, introduced with the keywords `rl` and `crl`. The mathematical variables in such statements are either explicitly declared with the keywords `var` and `vars`, or can be introduced on the fly in a statement without being declared previously, in which case they must have the form $var : sort$. Finally, a comment is preceded by '`***`' or '`---`' and lasts until the end of the line.

Maude modules are *executable* under reasonable assumptions. The high performance Maude engine—which can perform up to millions of rewrites per second—provides the following analysis commands:

– A *rewrite* (`rew`) and a *"fair" rewrite* (`frew`) command, which execute *one* rewrite sequence—out of possibly many—from a given initial state.
– A *search* command (`search`) for analyzing *all* possible rewrite sequences from a given initial state t_0, by performing a *breadth-first search* to check whether terms matching certain patterns can be reached from t_0. The search does not terminate if the set of states reachable from t_0 is infinite and the desired state(s) are not reachable from t_0.
– A *linear temporal logic model checker* [14], comparable to Spin [17] in performance, which checks whether each rewrite sequence from a given initial state t_0 satisfies a certain linear temporal logic (LTL) formula. LTL model checking will normally not terminate if the state space reachable from t_0 is infinite.
  A propositional LTL formula is constructed by the usual LTL operators (see, e.g., [10, 14] and Section 4.2.2) and a set Π of user-defined (possibly parametric) atomic propositions. Such atomic propositions should be defined as terms of the built-in sort `Prop`, in a module that includes the built-in Maude module `MODEL-CHECKER`. The labeling function L_Π is defined by equations of the form t `|=` p `=` b `if` C, for a (possibly) parametric atomic proposition p (i.e., for p a term of sort `Prop`), a term t of the built-in kind `[State]`, a term b of kind `[Bool]`, and a condition C. It is sufficient to define when a predicate *holds*. For example, if p were the only proposition, then $L_\Pi([u]) = \{\sigma(p) \mid \sigma \text{ ground substitution} \wedge (E \cup A) \vdash (\forall\emptyset)\, u \mathrel{\texttt{|=}} \sigma(p) = \texttt{true}\}$ [10].
– Finally, the user may define her own specific execution strategies using Maude's *reflective capabilities* [11,12].

[2] In Full Maude, and in its extension Real-Time Maude, module declarations and execution commands must be enclosed by a pair of parentheses.

[3] *Kinds* are not declared explicitly; the kind to which sort s belongs is written `[`s`]`.

We refer to the Maude manual [10] for a more thorough description of Maude's analysis capabilities.

2.2.1 Object-Oriented Specification in Maude

In object-oriented (Full) Maude[4] modules one can declare *classes* and *subclasses*. A class declaration

class C | att_1 : s_1 , ... , att_n : s_n .

declares an object class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term

$$< O : C \mid att_1 : val_1, ..., att_n : val_n >$$

of the built-in sort `Object`, where O is the object's name or identifier, and where val_1 to val_n are the current values of the attributes att_1 to att_n and have sorts s_1 to s_n. Objects can interact with each other in a variety of ways, including the sending of messages. A message is a term of the built-in sort `Msg`, where the declaration

msg m : $p_1 \dots p_n$ -> Msg .

defines the name of the message (m) and the sorts of its parameters ($p_1 \dots p_n$). In a concurrent object-oriented system, the state, which is usually called a *configuration* and is a term of the built-in sort `Configuration`, has typically the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative and having the `none` multiset as its identity element, so that order and parentheses do not matter, and so that rewriting is *multiset rewriting* supported directly in Maude. The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the configuration on the left-hand side of the rule

```
rl [l] :  m(O,w)  < O : C | a1 : x, a2 : y, a3 : z >  =>
                  < O : C | a1 : x + w, a2 : y, a3 : z >  m'(y,x) .
```

contains a message `m`, with parameters `O` and `w`, and an object `O` of class `C`. The message `m(O,w)` does not occur in the right-hand side of this rule, and can be considered to have been *removed* from the state by the rule. Likewise, the message `m'(y,x)` only occurs in the configuration on the right-hand side of the rule, and is thus *generated* by the rule. The above rule, therefore, defines a (parameterized family of) transition(s) in which a message `m(O,w)` is read, and consumed, by an object `O` of class `C`, with the effect of altering the attribute `a1` of the object and of sending a new message `m'(y,x)`. Attributes, such as `a3` in our example, whose values do not change and do not affect the next state of other attributes need not be mentioned in a rule. Attributes, like `a2`, whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, may be omitted from right-hand sides of rules. Thus the above rule could also be written

```
rl [l] :  m(O,w)  < O : C | a1 : x, a2 : y >  =>
                  < O : C | a1 : x + w >  m'(y,x) .
```

A *subclass* inherits all the attributes and rules of its superclasses[5].

[4] Real-Time Maude is built on top of Full Maude [10, Part II], which extends Maude with support for object-oriented specification and advanced module operations.

[5] The attributes and rules of a class cannot be modified by its subclasses, which may of course have additional attributes and rules.

3 Real-Time Rewrite Theories Revisited

In [28] we proposed to specify real-time and hybrid systems in rewriting logic as *real-time rewrite theories*, and defined an extension of the basic model to include the possibility of defining *eager* and *lazy* rewrite rules. This section first recalls the definition of real-time rewrite theories, and then explains why the generalization of rewriting logic given in [7] has made the partition into eager and lazy rules unnecessary.

3.1 Real-Time Rewrite Theories

A real-time rewrite theory is a rewrite theory where some rules, called *tick rules*, model time elapse in a system, while "ordinary" rewrite rules model instantaneous change.

Definition 1 A *real-time rewrite theory* $\mathscr{R}_{\phi,\tau}$ is a tuple $(\mathscr{R},\phi,\tau)$, where $\mathscr{R}=(\Sigma,E,\varphi,R)$ is a (generalized) rewrite theory, such that

- ϕ is an equational theory morphism $\phi : \mathit{TIME} \to (\Sigma,E)$ from the theory TIME to the underlying equational theory of $\mathscr{R}$, that is, ϕ interprets TIME in $\mathscr{R}$; the theory TIME [28] defines time abstractly as an ordered commutative monoid $(\mathit{Time},0,+,<)$ with additional operators such as $\dot{-}$ (where $x \dot{-} y$ denotes $x-y$ if $y<x$, and 0 otherwise) and $\leq$;
- (Σ,E) contains a sort `System` (denoting the state of the system), and a specific sort `GlobalSystem` with no subsorts or supersorts and with only one operator

$$\texttt{\{\_\}} : \texttt{System} \to \texttt{GlobalSystem}$$

  which satisfies no non-trivial[6] equations; furthermore, the sort `GlobalSystem` does not appear in the arity of any function symbol in Σ;
- τ is an assignment of a term τ_l of sort $\phi(\mathit{Time})$ to every rewrite rule

$$l : \texttt{\{}t\texttt{\}} \longrightarrow \texttt{\{}t'\texttt{\}} \ \textbf{if} \ \mathit{cond}$$

  involving terms of sort `GlobalSystem`[7]; if $\tau_l \neq \phi(0)$ we call the rule a *tick rule* and write

$$l : \ \texttt{\{}t\texttt{\}} \xrightarrow{\tau_l} \texttt{\{}t'\texttt{\}} \ \textbf{if} \ \mathit{cond}.$$

  The term τ_l denoting the *duration* of the tick rule may contain variables, including variables that do not occur in t, t', and/or cond. For example, if τ_l is a variable x not occurring in either t or cond, then time can advance nondeterministically by *any* amount from a substitution instance of $\texttt{\{}t\texttt{\}}$ where the substitution satisfies cond.

The global state of the system should have the form $\texttt{\{}u\texttt{\}}$, in which case the form of the tick rules ensures that time advances uniformly in all parts of the system. The total time elapse $\tau(\alpha)$ of a rewrite $\alpha : \texttt{\{}t\texttt{\}} \longrightarrow \texttt{\{}t'\texttt{\}}$ of sort `GlobalSystem` is the sum of the times elapsed in each tick rule application [28]. We write $\mathscr{R}_{\phi,\tau} \vdash \texttt{\{}t\texttt{\}} \xrightarrow{r} \texttt{\{}t'\texttt{\}}$ if there is a proof $\alpha : \texttt{\{}t\texttt{\}} \longrightarrow \texttt{\{}t'\texttt{\}}$ in $\mathscr{R}_{\phi,\tau}$ with $\tau(\alpha)=r$. Furthermore, we write $\mathit{Time}_\phi, 0_\phi, \ldots,$ for $\phi(\mathit{Time})$, $\phi(0)$, etc.

6 By "trivial" equations we mean equations of the form $t=t$.

7 All rules involving terms of sort `GlobalSystem` are assumed to have different labels.

3.2 Eager and Lazy Rules Revisited

The motivation behind having *eager* and *lazy* rewrite rules was to model *urgency* by letting the application of instantaneous eager rules take precedence over the application of lazy tick rules [28]. This feature was supported in version 1 of Real-Time Maude. The ability to define *frozen* operators in rewriting logic [7] means that it is no longer necessary to explicitly define eager and lazy rules. Instead, one may define a frozen operator[8]

$$eagerEnabled : s \to \texttt{[Bool] [frozen (1)]}$$

for each sort s that can be rewritten, introduce an equation

$$eagerEnabled(t) = \texttt{true} \textbf{ if } cond$$

for each "eager" rule $t \longrightarrow t'$ **if** $cond$, and add an equation

$$eagerEnabled(f(x_1, \ldots, x_n)) = \texttt{true} \textbf{ if } eagerEnabled(x_i) = \texttt{true}$$

for each operator f and each position i which is not a frozen position in f. A "lazy" tick rule should now have the form

$$l : \texttt{\{}t\texttt{\}} \xrightarrow{\tau_l} \texttt{\{}t'\texttt{\}} \textbf{ if } cond \wedge eagerEnabled(\texttt{\{}t\texttt{\}}) \neq \texttt{true}.$$

This technique makes unnecessary any explicit support for eager and lazy rules at the system definition level to model urgency. In addition, the lazy/eager feature has not been needed in any Real-Time Maude application we have developed so far. Real-Time Maude 2 therefore does not provide explicit support for defining eager and lazy rules.

4 Specification and Execution in Real-Time Maude

This section gives an overview of how to specify real-time rewrite theories in Real-Time Maude as *timed modules*, and how to execute such modules in the tool. In particular, Section 4.1.5 presents some useful techniques for specifying object-oriented real-time systems in Real-Time Maude. The manual [24] explains our tool in much more detail.

4.1 Specification in Real-Time Maude 2.1

Real-Time Maude extends Full Maude [10] to support the specification of real-time rewrite theories as *timed modules* and *object-oriented timed modules*. Such modules are entered at the user level by enclosing them in parentheses and including the module body between the keywords `tmod` and `endtm`, and between `tomod` and `endtom`, respectively. To state non-executable properties, Real-Time Maude allows the user to specify real-time extensions of abstract Full Maude *theories*. Since Real-Time Maude extends Full Maude, we can also define Full Maude modules in the tool. All the usual operations on modules provided by Full Maude are supported in Real-Time Maude.

[8] By '`[frozen (1)]`' we mean that the first (and in this case only) argument of the corresponding operator ($eagerEnabled$) cannot be rewritten (see Section 2.1). That is, even if t rewrites to u, it is *not* the case that $eagerEnabled(t)$ rewrites to $eagerEnabled(u)$.

4.1.1 Specifying the Time Domain

The equational theory morphism ϕ in a real-time rewrite theory $\mathscr{R}_{\phi,\tau}$ is not given explicitly at the specification level. Instead, by default, any timed module automatically imports the following functional module `TIME`[9]:

```
fmod TIME is
   sorts Time NzTime .  subsort NzTime < Time .
   op zero : -> Time .
   op _plus_ : Time Time -> Time [assoc comm prec 33 gather (E e)] .
   op _monus_ : Time Time -> Time [prec 33 gather (E e)] .
   ops _le_ _lt_ _ge_ _gt_ : Time Time -> Bool [prec 37] .
   eq zero plus R:Time = R:Time .
   eq R:Time le R':Time = (R:Time lt R':Time) or (R:Time == R':Time) .
   eq R:Time ge R':Time = R':Time le R:Time .
   eq R:Time gt R':Time = R':Time lt R:Time .
endfm
```

The morphism ϕ implicitly maps $Time$ to `Time`, 0 to `zero`, $\_+\_$ to `_plus_`, $\_\leq\_$ to `_le_`, etc. Even though Real-Time Maude assumes a fixed syntax for time operations, the tool does not build in a fixed model of time. In fact, the user has complete freedom to specify the desired data type of time values—which can be either discrete or dense and need not be linear—by specifying the data elements of sort `Time`, and by giving equations interpreting the constant `zero` and the operators `_plus_`, `_monus_`, and `_lt_`, so that the axioms of the theory $TIME$ [28] are satisfied. The predefined Real-Time Maude module `NAT-TIME-DOMAIN` defines the time domain to be the natural numbers as follows:

```
fmod NAT-TIME-DOMAIN is including LTIME .  protecting NAT .
  subsort Nat < Time .  subsort NzNat < NzTime .
  vars N N' : Nat .
  eq zero = 0 .
  eq N plus N' = N + N' .
  eq N monus N' = if N > N' then sd(N, N') else 0 fi .
  eq N lt N' = N < N' .
endfm
```

To have dense time, the user can import the predefined module `POSRAT-TIME- DOMAIN`, which defines the nonnegative rationals to be the time domain. The set of predefined modules in Real-Time Maude also includes a module `LTIME`, which assumes a linear time domain and defines the operators `max` and `min` on the time domain, and the modules `TIME-INF`, `LTIME-INF`, `NAT-TIME-DOMAIN- WITH-INF`, and `POSRAT-TIME-DOMAIN-WITH-INF` which extend the respective time domains with an "infinity" value `INF` in a supersort `TimeInf` of `Time`. Detailed specifications for all these time domains can be found in [24, Appendix A].

4.1.2 Tick Rules

A timed module automatically imports the module `TIMED-PRELUDE` which contains the declarations

```
sorts System GlobalSystem .
op {_} : System -> GlobalSystem [ctor] .
```

A conditional tick rule $l : \{t\} \xrightarrow{\tau_l} \{t'\}$ **if** $cond$ is written with syntax

[9] The operator attributes `prec` and `gather` deal with parsing; their meaning is explained in [10].

```
crl [l] : {t} => {t'} in time τ_l if cond .
```

and with similar syntax for unconditional rules.

We do not require time to advance beyond any time bound, or the specification to be "non-Zeno." However, it seems sensible to require that if time can advance by r `plus` r' time units from a state `{`t`}` in one application of a tick rule, then it should also be possible to advance time by r time units from the same state using the same tick rule. Tick rules should (in particular for dense time) typically have one of the forms

```
crl [l] : {t} => {t'} in time x if cond /\ x le u /\ cond' [nonexec] .    (†),
crl [l] : {t} => {t'} in time x if cond /\ x lt u /\ cond' [nonexec] .    (‡),
crl [l] : {t} => {t'} in time x if cond [nonexec] .                       (∗), or
 rl [l] : {t} => {t'} in time x [nonexec] .                               (§),
```

where x is a variable of sort `Time` (or of a subsort of `Time`) which does not occur in `{`t`}` and which is not initialized in the condition. The term u denotes the maximum amount by which time can advance in one tick step. Each variable in u should either occur in t or be instantiated in $cond$ by *matching equations* (see [10]). The (possibly empty) conditions $cond$ and $cond'$ should not further constrain x (except possibly by adding the condition x `=/= zero`). Tick rules in which the duration term contains a variable that does not occur in the rule's lefthand side and is not initialized by matching equations in the rule's condition are called *time-nondeterministic*. All other tick rules are called *time-deterministic* and can be used e.g. in discrete time domains.

Real-Time Maude assumes that tick rule applications in which time advances by `zero` do not change the state of the system. A tick rule is *admissible* if its `zero`-time applications do not change the state, and it is either a time-deterministic tick rule or a time-nondeterministic tick rule of any of the above forms—possibly with `le` and `lt` replaced by `<=` and `<` (in which case `le` and `<=`, and `lt` and `<`, should be equivalent on the time domain). The execution of admissible tick rules is supported by the Real-Time Maude tool. However, time-nondeterministic tick rules are not directly executable by the underlying Maude engine, since many choices are possible for instantiating the time variable x (that is why they are specified with the `nonexec` attribute, which tells Maude that these rules are not intended to be executed before they have been treated by Real-Time Maude). Real-Time Maude executes such rules using a *time sampling strategy* (see Sections 4.2.1 and 5.2) specified by the user.

4.1.3 Defining Initial States

For the purpose of conveniently defining initial states, Real-Time Maude allows the user to introduce operators of sort `GlobalSystem`. Each ground term of sort `GlobalSystem` must reduce to a term of the form `{`t`}` using the equations in the specification. The constant `initState` on page 30 is an example of an operator of sort `GlobalSystem` which reduces to a term of the desired form.

4.1.4 Timed Object-Oriented Modules

Maude's object model can be extended to the real-time setting by just adding a subsort declaration

```
subsort Configuration < System .
```

where Configuration is the sort whose elements are multisets of messages and objects. Timed object-oriented modules extend both object-oriented and timed modules to provide support for object-oriented real-time systems. In contrast to untimed object-oriented systems, functions such as δ and mte (described below), and the tick rules, will manipulate the global configuration. It is therefore useful to have a richer sort structure for configurations. Timed object-oriented modules include subsorts for nonempty configurations (NEConfiguration), configurations without messages (ObjectConfiguration) or without objects (MsgConfiguration), etc. Real-Time Maude automatically adds the subsort declaration Configuration < System to timed object-oriented modules. Section 6.2 gives an example of a timed object-oriented module.

4.1.5 Useful Techniques for Object-Oriented Specification in Real-Time Maude

In this section we present some techniques for specifying object-oriented systems in Real-Time Maude that have proved useful in all our larger case studies. These specification techniques provide a more elegant and natural way of specifying object-oriented systems than those given in [28]. This improvement is due to the possibility of having *frozen* operators in version 2 of Maude (and in Real-Time Maude).

In larger object-oriented systems it is usually the case that an unbounded number of objects could be affected by the elapse of time and/or could affect the maximum time elapse in a tick step. For such systems, we have found it useful to have functions

```
op δ : Configuration Time -> Configuration [frozen (1)] .
```

and

```
op mte : Configuration -> TimeInf [frozen (1)] .
```

to define, respectively, the effect of passage of time on a configuration, and the *m*aximum *t*ime *e*lapse possible from a configuration, and to let these functions distribute over the elements in a configuration according to the following equations:

```
vars NeC NeC' : NEConfiguration .     var R : Time .

eq δ(none, R) = none .
eq δ(NeC NeC', R) = δ(NeC, R) δ(NeC', R) .

eq mte(none) = INF .
eq mte(NeC NeC') = min(mte(NeC), mte(NeC')) .
```

The functions δ and mte must then be defined on *individual* objects and messages, as exemplified in Section 6.2.[10]

The tick rule(s)—there is usually just one tick rule—then typically have the form

```
crl [tick] :
    {SYSTEM:Configuration}
   =>
    {δ(SYSTEM:Configuration, R:Time)} in time R:Time
    if R:Time <= mte(SYSTEM:Configuration) [nonexec] .
```

The *instantaneous* rewrite rules, i.e., all rules except the tick rule(s), are defined exactly as in untimed rewriting logic.

[10] The functions δ and mte are *not* predefined in Real-Time Maude. They must be declared and defined by the user.

4.2 Formal Analysis in Real-Time Maude

Our tool translates a timed module into an untimed module which can be executed in Maude. However, the following reasons indicate that it is useful to go beyond Maude's standard rewriting, search, and model checking capabilities to execute and analyze timed modules:

- Tick rules are typically time-nondeterministic and cannot be executed directly in Maude.
- It is often more natural to measure and control the rewriting by the total duration of a computation than by the number of rewrites performed.
- Search and temporal logic properties often involve the duration of a computation (e.g., is a certain state always reached within time r? is there a potential deadlock in the time interval $[r, r')$?).
- One natural way of reducing the reachable state space from an infinite set to a finite set for model checking purposes is to consider only all behaviors *up to* a certain time bound r.

In Section 4.2.1 we describe the tool's *time sampling strategies*, which guide the application of time-nondeterministic tick rules. Section 4.2.2 gives an overview of the analysis commands available in Real-Time Maude. These commands are timed versions of Maude's rewriting, search, and model checking commands. To achieve high performance, our tool executes Real-Time Maude commands by transforming a timed module and command into an ordinary Maude module and command which is then executed in Maude as explained in Section 5.

4.2.1 Time Sampling Strategies

The issue of treating admissible time-nondeterministic tick rules is closely related to the treatment of dense time. The decidable timed automaton formalism [3] "discretizes" dense time by defining "clock regions," so that all states in the same clock region are bisimilar and satisfy the same properties [3]. The clock region construction is possible due to the restrictions in the timed automaton formalism, but in general it cannot be employed in the more complex systems expressible in Real-Time Maude. Our tool instead deals with admissible time-nondeterministic tick rules by offering a choice of different "time sampling" strategies, so that instead of covering the whole time domain, only *some* moments are visited.

The Real-Time Maude command

(set tick def r .)

for r a ground term of sort `Time` in the "current" module, sets the time sampling strategy to the *default* mode, which means that each application of a time-nondeterministic tick rule will try to advance time by r time units. (If the tick rule has the form (†), then the time advance is the minimum of u and r.) The command `(set tick max .)` can be used when all time-nondeterministic tick rules have the form (†) to set a time sampling strategy which advances time by the largest possible amount, namely u. The command (set tick max def r .) sets the time sampling strategy to advance time by the maximum possible time elapse u in rules of the form (†) (unless u equals `INF`), and tries to advance time by r time units in tick rules having other forms. The time sampling strategy stays unchanged until another strategy is selected by the user. Initially it is set to *deterministic* (`det`) mode, in which case it is assumed that all tick rules are time-deterministic.

All applications of time-nondeterministic tick rules—be it for rewriting, search, or model checking—are performed using the current time sampling strategy. This means that some

behaviors in the system, namely those obtained by applying the tick rules differently, are not analyzed. The results of Real-Time Maude analysis should be understood as being in general incomplete: counterexamples are true counterexamples, but (except for the case of discrete time when all states are visited) satisfaction of a property only shows that it holds for the states visited. We are currently working on identifying classes of real-time systems and system properties for which a given time sampling strategy actually preserves the relevant system properties and therefore provides a complete method of analysis.

4.2.2 Real-Time Maude Analysis

The *timed rewrite* command

(trew [n] in mod : t_0 in time <= r .)

simulates (at most n rewrite steps of) *one* behavior of the system, specified by the timed module mod, from initial state t_0 (of sort GlobalSystem) up to a total duration less than or equal to the Time value r. The time bound can also have the forms in time < r and with no time limit. The *timed fair rewrite* (tfrew) command applies the rules in a position-fair and rule-fair way. The '[n]' and 'in mod :' parts of the command are optional. Real-Time Maude's *tracing* facilities allow us to trace the steps in a timed rewrite sequence (see [24] for details).

The *timed search* command can be used to analyze not just *one* behavior, but to analyze *all* behaviors from a given initial state, relative to the chosen time sampling strategy. This command extends Maude's search command to search for states which match a *search pattern* and which are reachable in a given time interval. The syntax variations of the timed search command are:

(tsearch t_0 $arrow$ $pattern$ with no time limit .)
(tsearch t_0 $arrow$ $pattern$ in time $\sim$ r .)
(tsearch t_0 $arrow$ $pattern$ in time-interval between $\sim'$ r and $\sim''$ r' .)

where t_0 is a ground term of sort GlobalSystem, $pattern$ is either t or has the form t such that $cond$ for a ground irreducible[11] term t of sort GlobalSystem and a semantic condition $cond$ on the variables occurring in t, $\sim$ is either <, <=, >, or >=, $\sim'$ is either >= or >, $\sim''$ is either <= or <, and r and r' are ground terms of sort Time. The $arrow$ is the same as in Maude, where =>1, =>*, and =>+ search for states reachable from t_0 in, respectively, one, zero or more, and one or more rewrite steps. The arrow =>! is used to search for "deadlocked" states, i.e., states which cannot be further rewritten. The timed search command can be parameterized by the number of solutions sought and/or by the module to be analyzed.

Real-Time Maude also has commands which search for the *earliest* time and the *latest* time at which a state satisfying the desired $pattern$ can be reached. These commands are written with syntax

(find earliest t_0 =>* $pattern$.)
(find latest t_0 =>* $pattern$ $timeBound$.)

[11] A term t is *ground irreducible* if and only if for all ground substitutions σ such that, for each variable x, the ground term $\sigma(x)$ is irreducible (using the equations in the specification), then the term $\sigma(t)$ is itself irreducible.

We can also analyze all *behaviors* of a system from a given initial state, relative to the chosen time sampling strategy, using Real-Time Maude's *time-bounded explicit-state linear temporal logic model checker*. Such model checking extends Maude's high performance model checker [14] by analyzing the rewrite sequences only up to a given time bound. Temporal formulas are formed exactly as in Maude, that is, as terms of sort `Formula` constructed by user-defined atomic propositions and operators such as `/\` (conjunction), `\/` (disjunction), `->` (implication), `~` (negation), `[]` ("always"), `<>` ("eventually"), `U` ("until"), `=>` ("always implies"), etc. Atomic propositions, possibly parameterized, are terms of sort `Prop` and their semantics is defined by stating for which states a property holds. Propositions may be *clocked*, in that they also take the elapsed time into account. That is, whether a clocked proposition holds for a certain state depends not only on the state, but also on the total duration of the rewrite sequence leading up to the state. The proposition `clockEqualsTime` on page 27 shows an example of a clocked proposition. A module defining the propositions should import the predefined module `TIMED-MODEL-CHECKER` and the timed module to be analyzed. A formula represents an untimed linear temporal logic formula; it is *not* a formula in *metric temporal logic* or some other real-time temporal logic [4]. The syntax of the time-bounded model checking command is

(`mc` t_0 `|=t` $formula$ `in time <=` r `.`)

or with time bounds of the form `<` r or `with no time limit`. The model checker in general cannot *prove* a formula correct in the presence of time-nondeterministic tick rules, since it then only analyzes a subset of all possible behaviors. However, if the tool finds a counterexample, it is a valid counterexample which proves that the formula does not hold. *Time-bounded* model checking is guaranteed to terminate for discrete time domains when the instantaneous rules terminate.

The set of states reachable from an initial state in a timed module may well be finite, in which case search and model checking should terminate. However, the internal representation of a timed module described in Section 5 adds a clock component to each state, which makes the reachable "clocked state" space infinite, unless the specification is terminating. Real-Time Maude therefore also provides *untimed search* (syntax (`utsearch` t_0 $arrow$ $pattern$ `.`)) and *untimed model checking* (syntax (`mc` t_0 `|=u` $formula$ `.`)) where the internal representation used for the execution does not add a clock, and therefore preserves the finiteness of the reachable state space.

Real-Time Maude also has commands for checking "until" properties (syntax (`check` t_0 `|=` $pattern_1$ `until` $pattern_2$ $timeBound$ `.`)) and "until/stable" properties (syntax (`check` t_0 `|=` $pattern_1$ `untilStable` $pattern_2$ $timeBound$ `.`)). While the properties that can be expressed by these commands are a restricted (but often useful) subset of those expressible in temporal logic, the `check` commands are implemented using breadth-first search techniques, and can therefore sometimes decide properties—without restricting the duration of the behaviors—for which temporal logic model checking does not terminate.

Finally, the user can define his/her own specific analysis and verification strategies using Real-Time Maude's reflective capabilities to further analyze a timed module. The predefined module `TIMED-META-LEVEL` extends Maude's `META-LEVEL` module with the functionality needed to execute timed modules and can be used for these purposes.

4.3 Expressiveness and Limitations of Real-Time Maude

As mentioned in the introduction, our tool emphasizes ease and generality of specification, so that large and complex systems involving, e.g., different data types and forms of commu-

nication, can be modeled without having to resort to tricky encodings or imposing limitations on the system to be modeled. To support this claim, we showed in [28] that a wide range of models of real-time and hybrid systems, including timed [3] and hybrid automata [2], timed Petri nets [1], and timed and phase transition systems [21], can all be naturally expressed as real-time rewrite theories. In addition, Real-Time Maude supports the definition of any computable data type, as well as advanced object-oriented specification features such as multiple inheritance and creation/deletion of objects and messages. Real-Time Maude does not come with built-in communication primitives; instead, the user can define her own form(s) of communication at the desired level of abstraction, without having to encode them using a given set of basic primitives. This has allowed us to model unicast message passing with different transmission times (see, e.g., Section 6.2) and more advanced communication forms such as multicast (with appropriate transmission times) through links [29] and geographically bounded broadcast in wireless sensor network systems [30]. In terms of expressiveness, Real-Time Maude stands in stark contrast not only to the timed and hybrid automata, but also to other formalisms and tools, such as the real-time models mentioned above, network simulation tools, and the IF toolset [6]. Despite this flexibility, our formalism—consisting of equations and term rewrite rules—is simple and intuitive and has a well-defined and easy to understand semantics [28].

Given the expressiveness of Real-Time Maude, it is no surprise that most system properties are in general undecidable. This is different from, e.g., timed automata, whose formalism is restricted so that crucial properties remain decidable. Nevertheless, for discrete time—all our larger Real-Time Maude applications have had discrete time domain—Real-Time Maude search and LTL model checking can often be used to analyze all possible behaviors up to a given duration from a given initial state, thus becoming decision procedures. For dense time, however, our tool only offers a set of time sampling strategies, and, as mentioned in Section 4.2.1, there is no guarantee that Real-Time Maude search and model checking are "complete" in these cases. Such analyses cannot be used to prove that some property holds for all behaviors. They should instead be seen as analyzing a number of behaviors for the purpose of finding errors or to strengthen our confidence in the specification.

We can summarize the differences between Real-Time Maude and well-known timed automaton-based tools, such as UPPAAL [19,5] and Kronos [32], as follows: Many large and complex systems can be naturally modeled in Real-Time Maude but not in UPPAAL or Kronos. This applies to the Real-Time Maude applications listed in Section 6.4, but also to the smaller examples in Sections 6.2 and 6.3. In Section 6.2, there is no bound on the number of messages that can appear in the state, so this simple system cannot be modeled by a timed or a hybrid automaton. The example in Section 6.3 can be modeled by a hybrid automaton, but due to the uninitialized "stopwatch," it cannot be modeled within the *decidable* fragments of hybrid automata [16]. However, *when* timed automaton-based tools can be applied, they provide the following advantages over Real-Time Maude:

- Model checking[12] of timed automata is guaranteed to terminate, while the corresponding Maude analysis may fail to do so.
- UPPAAL, in particular, is a very efficient model checking tool for timed automata, where sets of clock valuations are represented symbolically. Real-Time Maude, which is not optimized for the special case of timed automata, uses explicit-state search and model checking.

[12] UPPAAL's query language is only a limited subset of (untimed) CTL [5] while Real-Time Maude allows us to define any propositional linear temporal logic formula. Kronos' query language is *timed* CTL (TCTL) [4].

– Model checking of timed automata is complete also for dense time.

5 Semantics of Real-Time Maude's Analysis Commands

Real-Time Maude is designed to take maximum advantage of the high performance of the Maude engine. Most Real-Time Maude analysis commands are therefore executed by first transforming the current timed module into a Maude module, followed by the execution of a corresponding Maude command (at the Maude *meta-level*). The actual transformation of a timed module depends on the Real-Time Maude command to execute. This section defines the semantics of Real-Time Maude's analysis commands in two ways by providing:

– an "abstract" semantics, which specifies requirements for each command; and
– a concrete "Maude semantics," which defines the semantics of a Real-Time Maude command as the theory transformation and Maude command used to execute it.

In what follows we show how the concrete semantics satisfies the abstract one. The concrete "Maude semantics" adopts a *reductionistic* approach based on semantics-preserving theory transformations. As explained in Section 5.1, any real-time rewrite theory can be transformed into a semantically equivalent ordinary rewrite theory. This fact is systematically exploited in our concrete "Maude semantics," to internally transform real-time commands into ordinary Maude commands. The subtle point, however, is that, as we explain for each command, the Real-Time Maude module and command must be transformed *together* into a corresponding Maude module and command. This is because the command itself places additional constraints, due to, e.g., the specified time bound or the time sampling strategy, that must be reflected in the transformed theory. For example, the transformed tick rule should not tick the time beyond the time bound specified in the command.

Section 5.1 describes the "default" transformation of a real-time rewrite theory into an ordinary rewrite theory, and therefore of Real-Time Maude modules into Maude modules. Section 5.2 gives the semantics of the time sampling strategies. Sections 5.4 to 5.6 present the semantics of, respectively, the timed rewrite commands, timed search and related commands, and time-bounded linear temporal logic model checking. Section 5.7 treats Real-Time Maude's *untimed* analysis commands.

5.1 The Clocked Transformation

Definition 2 The *clocked transformation*, which maps a real-time rewrite theory $\mathscr{R}_{\phi,\tau}$ with $\mathscr{R} = (\Sigma, E, \varphi, R)$ to an ordinary rewrite theory $(\mathscr{R}_{\phi,\tau})^C = (\Sigma^C, E^C, \varphi^C, R^C)$, adds the declarations

```
sorts ClockedSystem .
subsort GlobalSystem < ClockedSystem .
op _in time_ : GlobalSystem Time_φ -> ClockedSystem  [ctor] .
eq (CLS:ClockedSystem in time R:Time_φ) in time R':Time_φ =
       CLS:ClockedSystem in time (R:Time_φ +_φ R':Time_φ) .
```

to (Σ, E, φ), and defines R^C to be the union of the instantaneous rules in R and a rule

$$l : \{t\} \longrightarrow \{t'\}\ \texttt{in time}\ \tau_l\ \textbf{if}\ cond$$

for each corresponding tick rule $l : \{t\} \xrightarrow{\tau_l} \{t'\}$ **if** $cond$ in R.

This clocked transformation adds a clock component to each state and resembles the transformation $(\_)^C$ described in [28], but is simpler, since it is essentially the identity. It is worth noticing that the reachable state space from a state $\{t\}$ in $(\mathscr{R}_{\phi,\tau})^C$ is normally infinite, even when the reachable state space from $\{t\}$ is finite in $\mathscr{R}_{\phi,\tau}$. The arguments in [28] can easily be adapted to show:

Fact 1 *For all terms t, t' of sort* `GlobalSystem` *and all terms $r \neq 0_\phi$, r' of sort $Time_\phi$ in $\mathscr{R}_{\phi,\tau}$,*

- $\mathscr{R}_{\phi,\tau} \vdash t \xrightarrow{r} t' \iff (\mathscr{R}_{\phi,\tau})^C \vdash t \longrightarrow t'$ `in time` r
  $\iff (\mathscr{R}_{\phi,\tau})^C \vdash t$ `in time` $r' \longrightarrow t'$ `in time` $r' +_\phi r$, *and*
- $\mathscr{R}_{\phi,\tau} \vdash t \xrightarrow{0_\phi} t' \iff (\mathscr{R}_{\phi,\tau})^C \vdash t$ `in time` $r' \longrightarrow t'$ `in time` r'.

In addition, $\mathscr{R}_{\phi,\tau} \vdash t \xrightarrow{0_\phi} t' \iff (\mathscr{R}_{\phi,\tau})^C \vdash t \longrightarrow t'$ holds when $\mathscr{R}_{\phi,\tau}$ contains only admissible tick rules. Moreover, these equivalences hold for n-step rewrites for all n.

In Real-Time Maude, thanks to its syntax, this transformation is performed by importing the module `TIMED-PRELUDE`, which contains the above declarations (with `Time` for $Time_\phi$, etc.), and by *leaving the rest of the specification unchanged*. Real-Time Maude internally stores a timed module by means of its clocked representation. All Full Maude commands extend to Real-Time Maude and execute this clocked representation of the current timed module. Fact 1 justifies this choice of execution.

5.2 Time Sampling Strategies

Definition 3 The set $tss(\mathscr{R}_{\phi,\tau})$ of *time sampling strategies* associated with the real-time rewrite theory $\mathscr{R}_{\phi,\tau}$ with $\mathscr{R} = (\Sigma, E, \varphi, R)$ is defined by

$$tss(\mathscr{R}_{\phi,\tau}) = \{def(r) \mid r \in \mathbb{T}_{\Sigma,Time_\phi}\} \cup \{max\} \cup \{maxDef(r) \mid r \in \mathbb{T}_{\Sigma,Time_\phi}\} \cup \{det\}.$$

In Real-Time Maude, these time sampling strategies are "set" with the respective commands `(set tick def` r `.)`, `(set tick max .)`, `(set tick max def` r `.)`, and `(set tick det .)`.

Definition 4 For each $s \in tss(\mathscr{R}_{\phi,\tau})$, the mapping which takes the real-time rewrite theory $\mathscr{R}_{\phi,\tau}$ to the real-time rewrite theory $\mathscr{R}^s_{\phi,\tau}$, in which the admissible time-nondeterministic tick rules are applied according to the time sampling strategy s, is defined as follows:

- $\mathscr{R}^{def(r)}_{\phi,\tau}$ equals $\mathscr{R}_{\phi,\tau}$, with the admissible time-nondeterministic tick rules of the forms (†), (‡), ($*$), and (§) in Section 4.1.2 replaced by, respectively, the following tick rules[13]:
  - $l : \{t\} \xrightarrow{x} \{t'\}$ **if** $cond \wedge x :=$ `if` $(u \leq_\phi r)$ `then` u `else` r `fi` $\wedge\ x \leq_\phi u \wedge cond'$
  - $l : \{t\} \xrightarrow{x} \{t'\}$ **if** $x := r \wedge cond \wedge x <_\phi u \wedge cond'$
  - $l : \{t\} \xrightarrow{x} \{t'\}$ **if** $x := r \wedge cond$
  - $l : \{t\} \xrightarrow{x} \{t'\}$ **if** $x := r$

[13] The Real-Time Maude tool assumes the modified tick rules to be executable, and therefore "removes" their `nonexec` attributes. The syntax $v := w$ is that of Maude for "matching equations" [10], where the ground-irreducible pattern v (in the above rules v is just the variable x) is matched against the result of evaluating w.

If the time domain is linear, so that ϕ can be extended to the theory $LTIME$ [28], the first of the above rules can be given in the simpler form

$$l : \texttt{\{}t\texttt{\}} \xrightarrow{x} \texttt{\{}t'\texttt{\}} \textbf{ if } cond \wedge x := min_\phi(u, r) \wedge cond'.$$

– $\mathscr{R}^{max}_{\phi,\tau}$ is $\mathscr{R}_{\phi,\tau}$ with each rule of the form (†) replaced by the rule

$$l : \texttt{\{}t\texttt{\}} \xrightarrow{x} \texttt{\{}t'\texttt{\}} \textbf{ if } cond \wedge x := u \wedge cond'$$

(and with the other tick rules left unchanged). Notice that the condition does not hold if u evaluates to the infinity value.

– $\mathscr{R}^{maxDef(r)}_{\phi,\tau}$ equals $\mathscr{R}^{def(r)}_{\phi,\tau}$ with each (†)-rule replaced by the rule

$$l : \texttt{\{}t\texttt{\}} \xrightarrow{x} \texttt{\{}t'\texttt{\}} \textbf{ if } cond \wedge x := \texttt{if } u : Time_\phi \texttt{ then } u \texttt{ else } r \texttt{ fi} \wedge x \leq_\phi u \wedge cond'.$$

– $\mathscr{R}^{det}_{\phi,\tau} = \mathscr{R}_{\phi,\tau}$.

Real-Time Maude implements these transformations, with `le` for $\leq_\phi$, etc. We do not assume that the time domain is linear. By the *current* time sampling strategy we mean the time sampling strategy defined by the *last* `set tick` command given, and we assume that any time value used in the last `set tick` command is a time value in the "current" module.

The set of rewrites using a particular time sampling strategy is a subset of all possible rewrites:

Fact 2 *For each $s \in tss(\mathscr{R}_{\phi,\tau})$, $\mathscr{R}^{s}_{\phi,\tau} \vdash t \xrightarrow{r} t'$ implies $\mathscr{R}_{\phi,\tau} \vdash t \xrightarrow{r} t'$ for all terms t, t' of sort* `GlobalSystem`*, and all ground terms r of sort $Time_\phi$. Furthermore, this property holds for all n-step rewrites.*

5.3 Tick Rules with `zero` Time Advance

Real-Time Maude does not apply a tick rule when time would advance by an amount equal to `zero`. This is a pragmatic choice based on the fact that advancing time by `zero` using admissible tick rules does not change the state, but leads to unnecessary looping during executions. We denote by $\mathscr{R}^{nz}_{\phi,\tau}$ the real-time rewrite theory obtained from $\mathscr{R}_{\phi,\tau}$ by adding the condition $\tau_l \neq 0_\phi$ to each tick rule. We write $\mathscr{R}^{s,nz}_{\phi,\tau}$ for $(\mathscr{R}^{s}_{\phi,\tau})^{nz}$.

Fact 3 *$\mathscr{R}^{nz}_{\phi,\tau} \vdash t \xrightarrow{r} t'$ implies $\mathscr{R}_{\phi,\tau} \vdash t \xrightarrow{r} t'$. The implication extends to rewrites of length n for any n, and is an equivalence for specifications $\mathscr{R}_{\phi,\tau}$ with only admissible tick rules.*

5.4 Timed Rewriting

The timed rewrite command

(`trew [`n`] in` $\mathscr{R}_{\phi,\tau}$ `:` t `with no time limit .`),

for t a term of sort `GlobalSystem`, returns a term t' such that

– $\mathscr{R}_{\phi,\tau} \vdash t \longrightarrow t'$ is a rewrite in at most n steps, and
– t' cannot be further rewritten in $\mathscr{R}^{s,nz}_{\phi,\tau}$ (for s the current time sampling strategy) unless $t \longrightarrow t'$ is a rewrite in exactly n steps.

This command is executed at the Maude meta-level by (a call to a built-in function equivalent to) executing the Maude command

$\texttt{rewrite [}n\texttt{] in } (\mathscr{R}_{\phi,\tau}^{s,nz})^C \texttt{ : } t \texttt{ .}$

for s the current time sampling strategy. The correctness of executing the timed command in this way follows from the fact that if the result is a term t' `in time` r, then $(\mathscr{R}_{\phi,\tau}^{s,nz})^C \vdash t \longrightarrow t'$ `in time` r, and we have $(\mathscr{R}_{\phi,\tau}^{s,nz})^C \vdash t \longrightarrow t'$ `in time` $r \Longrightarrow \mathscr{R}_{\phi,\tau}^{s,nz} \vdash t \xrightarrow{r} t'$ $\Longrightarrow \mathscr{R}_{\phi,\tau} \vdash t \xrightarrow{r} t'$. All implications preserve the number of rewrite steps. Finally, it also follows from Fact 1 that t' cannot be rewritten further in $\mathscr{R}_{\phi,\tau}^{s,nz}$ if t' `in time` r cannot be rewritten in $(\mathscr{R}_{\phi,\tau}^{s,nz})^C$. The correctness argument is analogous if the result of the rewrite command is a `GlobalSystem` term t'.

Let $\sim$ stand for either `<=` or `<`, and let `<=`$_\phi$ and `<`$_\phi$ stand for $\leq_\phi$ and $<_\phi$. The time-bounded rewrite command

$\texttt{(trew [}n\texttt{] in } \mathscr{R}_{\phi,\tau} \texttt{ : } t \texttt{ in time } \sim r \texttt{ .)}$,

again for t a term of sort `GlobalSystem`, returns a term t' such that

– $\mathscr{R}_{\phi,\tau} \vdash t \xrightarrow{r'} t'$, for $r' \sim_\phi r$, is a rewrite in at most n steps, and
– either $t \xrightarrow{r'} t'$ is an n-step rewrite, or there is no t'' such that $\mathscr{R}_{\phi,\tau}^{s,nz} \vdash t' \xrightarrow{r''} t''$ for $r' +_\phi r'' \sim_\phi r$.

To execute time-bounded rewrite commands we use a different transformation of a real-time rewrite theory which ensures that the clocks associated to the states never go beyond the time limit.

Definition 5 Let $\mathscr{R}_{\phi,\tau}$ be a real-time rewrite theory with $\mathscr{R} = (\Sigma, E, \varphi, R)$, and let $r \in \mathbb{T}_{\Sigma,Time_\phi}$. The mapping which takes $\mathscr{R}_{\phi,\tau}$ to the rewrite theory $(\mathscr{R}_{\phi,\tau})^{\leq r} = (\Sigma^B, E^B, \varphi^B, R^{\leq r})$ is defined as follows:

– $\Sigma^B = \Sigma^C \cup \{\, \texttt{[\_]} : \texttt{ClockedSystem} \to \texttt{ClockedSystem}\,\}$[14],
– $E^B = E^C$,
– φ^B extends φ so that $\varphi^B(\texttt{[\_]}) = \emptyset$, and
– $R^{\leq r}$ is the union of the instantaneous rules in $\mathscr{R}_{\phi,\tau}$ and a rule

$$l : \texttt{[\{}t\texttt{\} in time } y\texttt{]} \longrightarrow \texttt{[\{}t'\texttt{\} in time } \tau_l +_\phi y\texttt{]} \textbf{ if } cond \wedge \tau_l +_\phi y \leq_\phi r$$

for each tick rule $l : \texttt{\{}t\texttt{\}} \xrightarrow{\tau_l} \texttt{\{}t'\texttt{\}}$ **if** $cond$ in $\mathscr{R}_{\phi,\tau}$, where y is a variable of sort $Time_\phi$ which does not occur in the original tick rule.

Fact 4

– *For all r', r'' with $r'' +_\phi r' \leq_\phi r$, we have that $\mathscr{R}_{\phi,\tau} \vdash t \xrightarrow{r'} t'$ if and only if $(\mathscr{R}_{\phi,\tau})^{\leq r} \vdash$ $\texttt{[}t \texttt{ in time } r''\texttt{]} \longrightarrow \texttt{[}t' \texttt{ in time } r'' +_\phi r'\texttt{]}$. In addition, the number of rewrite steps are the same in both sides of the equivalence.*
– *$(\mathscr{R}_{\phi,\tau})^{\leq r} \vdash \texttt{[}t \texttt{ in time } r'\texttt{]} \longrightarrow t''$ and $r' \leq_\phi r$ implies that t'' is a term of the form $\texttt{[}t' \texttt{ in time } r''\texttt{]}$ with $r'' \leq_\phi r$. That is, it is not possible to rewrite beyond the time limit.*

[14] The operator `[_]` is called `global` in the current implementation of the tool.

Real-Time Maude executes the time-bounded rewrite command

(trew [n] in $\mathscr{R}_{\phi,\tau}$: t in time <= r .)

by executing the command rewrite [n] in $(\mathscr{R}^{s,nz}_{\phi,\tau})^{\leq r}$: [t in time 0_ϕ] . in Maude.

For the correctness argument, it follows from Fact 4 that the result is [t' in time r'] for some $r' \leq_\phi r$ since $0_\phi \leq_\phi r$. By the first part of that fact, it follows that (since $r' = 0_\phi +_\phi r'$) $\mathscr{R}^{s,nz}_{\phi,\tau} \vdash t \xrightarrow{r'} t'$, which implies $\mathscr{R}_{\phi,\tau} \vdash t \xrightarrow{r'} t'$. Finally, it also follows from Fact 4 that there is no nontrivial rewrite $t' \xrightarrow{r''} t''$ with $r' +_\phi r'' \leq_\phi r$ in $\mathscr{R}^{s,nz}_{\phi,\tau}$ if [t' in time r'] cannot be further rewritten in $(\mathscr{R}^{s,nz}_{\phi,\tau})^{\leq r}$.

The execution of a timed rewrite command with a time bound of the form < r is entirely analogous, with each occurrence of the symbol $\leq$ replaced by the symbol $<$.

5.5 Timed Search

The timed search command

(tsearch [n] in $\mathscr{R}_{\phi,\tau}$: t_0 =>* t such that $cond$
        in time-interval between $\sim$ r and $\sim'$ r' .)

should return at most n substitutions σ satisfying $cond$ such that $\mathscr{R}_{\phi,\tau} \vdash t_0 \xrightarrow{r''} \sigma(t)$ for $r'' \sim_\phi r$ and $r'' \sim'_\phi r'$. It is executed as the Maude command

search [n] in $(\mathscr{R}^{s,nz}_{\phi,\tau})^{\sim' r'}$:
        [t_0 in time 0_ϕ] =>* [t in time TIME-ELAPSED]
                such that $cond$ /\ TIME-ELAPSED $\sim_\phi$ r .

for s the current time sampling strategy, and TIME-ELAPSED a variable of sort $Time_\phi$ which does not occur in t (otherwise a variable TIME-ELAPSED#1 is used).

For correctness, if σ is a solution, then $(\mathscr{R}^{s,nz}_{\phi,\tau})^{\sim' r'} \vdash$ [t_0 in time 0_ϕ] $\longrightarrow$ [$\sigma(t)$ in time σ(TIME-ELAPSED)]. By Fact 4, $\sigma(\texttt{TIME-ELAPSED}) \sim'_\phi r$ and $\mathscr{R}^{s,nz}_{\phi,\tau} \vdash t_0 \xrightarrow{\sigma(\texttt{TIME-ELAPSED})} \sigma(t)$, and therefore $\mathscr{R}_{\phi,\tau} \vdash t_0 \xrightarrow{\sigma(\texttt{TIME-ELAPSED})} \sigma(t)$. Finally, the such that condition implies that $\sigma(\texttt{TIME-ELAPSED}) \sim_\phi r$.

Real-Time Maude allows the term t in the search pattern to have the form t' in time t'', which is useful for searching for states matching patterns such as $t(x)$ in time x. Such patterns are treated by replacing TIME-ELAPSED with t''.

Since all the facts used in the argumentation preserve the number of rewrite steps, the same translation can be used with the arrows =>1 and =>+ instead of =>*.

It is worth remarking that

- the search will return (at most) n substitutions on the domain $vars(t) \cup \{\texttt{TIME-ELAPSED}\}$, which do not necessarily correspond to n *distinct* substitutions when restricted to $vars(t)$;
- the search will terminate if the time domain is discrete (or the time sampling strategy s makes $\mathscr{R}^{s,nz}_{\phi,\tau}$ "non-Zeno"), and the instantaneous rules terminate;
- solutions σ with $\mathscr{R}_{\phi,\tau} \vdash t_0 \xrightarrow{r''} \sigma(t)$ can be missed because it may be that $\mathscr{R}^{s,nz}_{\phi,\tau} \not\vdash t_0 \xrightarrow{r''} \sigma(t)$.

The time-bounded search command for deadlocks

```
(tsearch [n] in ℛ_{φ,τ} : t_0 =>! t  such that cond
         in time-interval between ∼ r and ∼' r' .)
```

searches for substitutions σ satisfying $cond$ such that $\mathscr{R}_{\phi,\tau} \vdash t_0 \xrightarrow{r''} \sigma(t)$ for $r'' \sim_\phi r$ and $r'' \sim'_\phi r'$, and such that $\sigma(t)$ cannot be further rewritten in $\mathscr{R}^{s,nz}_{\phi,\tau}$. The translation $(\mathscr{R}^{s,nz}_{\phi,\tau})^{\sim' r'}$ cannot be used since it would give deadlocks at all states which cannot be further rewritten *within the time bound*.

The following translation is used instead for searching for deadlocks. It adds a self-loop whenever a tick rule could advance the total time elapse of a computation beyond the time limit.

Definition 6 Let $\mathscr{R}_{\phi,\tau}$ be a real-time rewrite theory with $\mathscr{R} = (\Sigma, E, \varphi, R)$, and let $r \in \mathbb{T}_{\Sigma,Time_\phi}$. The mapping which takes $\mathscr{R}_{\phi,\tau}$ to the rewrite theory $(\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}}$ is defined by $(\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}} = (\Sigma^B, E^B, \varphi^B, R^{\widehat{\leq r}})$, where $R^{\widehat{\leq r}}$ is the union of the instantaneous rules in $\mathscr{R}_{\phi,\tau}$ and a rule

$$l : \texttt{[\{}t\texttt{\} in time } y\texttt{]} \longrightarrow \texttt{if } (\tau_l +_\phi y \leq_\phi r) \texttt{ then [\{}t'\texttt{\} in time } \tau_l +_\phi y\texttt{]} \\ \texttt{else [\{}t\texttt{\} in time } y\texttt{] fi} \quad \textbf{if } cond$$

for each tick rule $l : \texttt{\{}t\texttt{\}} \xrightarrow{\tau_l} \texttt{\{}t'\texttt{\}}$ **if** $cond$ in $\mathscr{R}_{\phi,\tau}$, where y is a variable of sort $Time_\phi$ which does not occur in the original tick rule.

The transformation $(\mathscr{R}_{\phi,\tau})^{\widehat{<r}}$ is defined in the same way.

Since $(\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}}$ only modifies $(\mathscr{R}_{\phi,\tau})^{\leq r}$ by adding trivial rewrites, most of Fact 4 also holds in $(\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}}$. Moreover, since the instantaneous rules are unchanged, and since for each tick rule which can be applied in $\mathscr{R}_{\phi,\tau}$, the corresponding rule can be applied to a corresponding state in $(\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}}$, it follows that a term can be rewritten in $\mathscr{R}_{\phi,\tau}$ if and only if it can be rewritten in $(\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}}$:

Fact 5

- *For all r', r'' with $r'' +_\phi r' \leq_\phi r$ it is the case that $\mathscr{R}_{\phi,\tau} \vdash t \xrightarrow{r'} t'$ if and only if $(\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}} \vdash$ $\texttt{[}t\texttt{ in time } r''\texttt{]} \longrightarrow \texttt{[}t'\texttt{ in time } r'' +_\phi r'\texttt{]}$. In addition, the number of rewrite steps can be preserved by the translation.*
- *$(\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}} \vdash \texttt{[}t\texttt{ in time } r'\texttt{]} \longrightarrow t''$ and $r' \leq_\phi r$ imply that t'' is (equivalent to) a term of the form $\texttt{[}t'\texttt{ in time } r''\texttt{]}$ with $r'' \leq_\phi r$. That is, it is not possible to rewrite beyond the time limit.*
- *If $\mathscr{R}_{\phi,\tau} \vdash t \xrightarrow{r'} t'$ is a one-step rewrite, and $r'' \leq_\phi r$ and $\neg(r'' +_\phi r' \leq_\phi r)$, then there is a one-step "identity" rewrite $(\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}} \vdash \texttt{[}t\texttt{ in time } r''\texttt{]} \longrightarrow \texttt{[}t\texttt{ in time } r''\texttt{]}$.*

The above timed search command for deadlocks is interpreted by the Maude command

```
search [n] in (ℛ^{s,nz}_{φ,τ})^{∼'r'} :
       [t_0 in time 0_φ] =>! [t in time TIME-ELAPSED]
              such that cond /\ TIME-ELAPSED ∼_φ  r .
```

To see that each solution σ is really a deadlock in $\mathscr{R}^{s,nz}_{\phi,\tau}$, assume that $\mathscr{R}^{s,nz}_{\phi,\tau} \vdash \sigma(t) \xrightarrow{r} t'$ in one step. It follows from Fact 5 that, depending on whether $r'' +_\phi r' \leq_\phi r$, the term $\texttt{[}\sigma(t)\texttt{ in time } r''\texttt{]}$ rewrites either to $\texttt{[}t'\texttt{ in time } r'' +_\phi r'\texttt{]}$ or to $\texttt{[}\sigma(t)\texttt{ in time } r''\texttt{]}$ in one step in $(\mathscr{R}^{s,nz}_{\phi,\tau})^{\widehat{\sim' r'}}$.

It is worth noticing that a deadlock in $\mathscr{R}^{s,nz}_{\phi,\tau}$ does not necessarily correspond to a deadlock in $\mathscr{R}_{\phi,\tau}$, and that a deadlock in $\mathscr{R}_{\phi,\tau}$ may not necessarily be reached in $\mathscr{R}^{s,nz}_{\phi,\tau}$.

For search commands with simpler time bounds, a command `(tsearch` t_0 *arrow* t `such that` *cond* `in time` $\sim$ r `.)` is equivalent to `(tsearch` t_0 *arrow* t `such that` *cond* `in time-interval between >=` 0_ϕ `and` $\sim$ r `.)` for $\sim$ either `<=` or `<`. If $\sim$ is either `>=` or `>`, the above search command is interpreted by the Maude command

$$\texttt{search [}n\texttt{] in } (\mathscr{R}^{s,nz}_{\phi,\tau})^C \texttt{ : } t_0 \ \mathit{arrow} \ t \texttt{ in time TIME-ELAPSED}$$
$$\texttt{such that } \mathit{cond} \texttt{ /\textbackslash{} TIME-ELAPSED } \sim_\phi r \texttt{ .}$$

A timed search command with bound '`with no time limit`' is the same as the corresponding search command with time bound `>=` 0_ϕ.

5.6 Time-Bounded Temporal Logic Model Checking

What is the meaning of the time-bounded liveness property "the `clock` value will always reach the value 24 within time 24" in the following specification?

```
(tmod CLOCK is protecting POSRAT-TIME-DOMAIN .
  op clock : Time -> System [ctor] .
  vars R R' : Time .
  rl [tick] : {clock(R)} => {clock(R + R')} in time R' [nonexec] .
endtm)
```

Real-Time Maude does *not* assume that time 24 must be "visited" when model checking a property "within time 24." Such an assumption would make the above property hold within time 24 but not within time 25, and an ordinary simulation would not necessarily reach the desired state, which is counterintuitive if we have proved that the desired state is always reached within time 24. Instead, time-bounded linear temporal logic formulas will be interpreted over all possible paths, "chopped off" at the time limit:

Definition 7 Given a real-time rewrite theory $\mathscr{R}_{\phi,\tau}$, a term t_0 of sort `GlobalSystem`, and a ground term r of sort Time_ϕ, the set $\mathit{Paths}(\mathscr{R}_{\phi,\tau})^{\leq r}_{t_0}$ is the set of *all* infinite sequences

$$\pi = ([t_0 \texttt{ in time } r_0] \longrightarrow [t_1 \texttt{ in time } r_1] \longrightarrow \cdots \longrightarrow [t_i \texttt{ in time } r_i] \longrightarrow \cdots)$$

of $(\mathscr{R}_{\phi,\tau})^C$-states, with $r_0 = 0_\phi$, such that either

- for all i, $r_i \leq_\phi r$ and $\mathscr{R}_{\phi,\tau} \vdash t_i \xrightarrow{r'} t_{i+1}$ is a one-step sequential rewrite for $r_i +_\phi r' = r_{i+1}$, or
- there exists a k such that
  - either there is a one-step rewrite $\mathscr{R}_{\phi,\tau} \vdash t_k \xrightarrow{r'} t'$ with $r_k \leq_\phi r$ and $r_k +_\phi r' \not\leq_\phi r$, or
  - there is no one-step rewrite from t_k in $\mathscr{R}_{\tau,\phi}$,

  and $\mathscr{R}_{\phi,\tau} \vdash t_i \xrightarrow{r'} t_{i+1}$ is a one-step sequential rewrite with $r_i +_\phi r' = r_{i+1}$ for all $i < k$; and $r_j = r_k$ and $t_j = t_k$ for all $j > k$.

We denote by $\pi(i)$ the ith element of path π.

That is, we add a self-loop for each deadlocked state reachable within time r, as well as for each state which *could* tick beyond time r in one step, even when it could *also* rewrite to something else within the time limit.

The temporal logic properties are given as ordinary LTL formulas over a set of atomic propositions. We find it useful to allow both *state propositions*, which are defined on terms of sort `GlobalSystem`, and *clocked propositions*, which can also take the time stamps into account. To allow clocked propositions, propositions are defined w.r.t. the *clocked* representation $(\mathscr{R}_{\phi,\tau})^C$ of a real-time rewrite theory $\mathscr{R}_{\phi,\tau}$. The satisfaction of a *state* proposition $\rho \in \Pi$ is independent of the time stamps, so the labeling function L_Π is extended to a labeling L_Π^C which is the "smallest" function satisfying $L_\Pi([t]) \subseteq L_\Pi^C([t])$ and $L_\Pi([t']) \subseteq L_\Pi^C([t' \texttt{ in time } r])$ for all t, t', and r.

In Real-Time Maude, we declare the atomic (state and clocked) propositions Π (as terms of sort `Prop`), and define their semantics L_Π, in a module which imports the module to be analyzed (represented by its clocked version) and the predefined module `TIMED-MODEL-CHECKER`. The latter extends Maude's `MODEL-CHECKER` module with the subsort declaration

```
subsort ClockedSystem < State .
```

Real-Time Maude transforms a module $\mathtt{M}_{L_\Pi}$ defining Π and L_Π into a module $\mathtt{M}_{L_\Pi^C}$ defining the labeling function L_Π^C by adding the conditional equation

```
ceq GS:GlobalSystem in time R:Time  |=  P:Prop = true
                if GS:GlobalSystem  |=  P:Prop .
```

The definition of the satisfaction relation of time-bounded temporal logic is given as follows:

Definition 8 Given a real-time rewrite theory $\mathscr{R}_{\phi,\tau}$, a protecting extension L_Π of $(\mathscr{R}_{\phi,\tau})^C$ defining the atomic state and clocked propositions Π, an initial state t_0 of sort `GlobalSystem`, a $Time_\phi$ value r, and an LTL formula Φ, we define the time-bounded satisfaction relation $\models_{\leq r}$ by

$$\mathscr{R}_{\phi,\tau}, L_\Pi, t_0 \models_{\leq r} \Phi \quad \text{if and only if} \quad \pi, L_\Pi^C \models \Phi \text{ for all paths } \pi \in Paths(\mathscr{R}_{\tau,\phi})_{t_0}^{\leq r},$$

where $\models$ is the usual definition of temporal satisfaction on infinite paths.

A time-bounded property which holds when a time sampling strategy is taken into account does not necessarily hold in the original theory. But a counterexample to a time-bounded formula when the time sampling strategy is taken into account, is also a valid counterexample in the original system if the time sampling strategy is different from det and all time-nondeterministic tick rules have the form (†):

Fact 6 *Let $\mathscr{R}_{\phi,\tau}$ be an admissible real-time rewrite theory where each time-nondeterministic tick rule has the form (†) with u a term of sort $Time_\phi$. Then, for any $Time_\phi$ value r, term t of sort* `GlobalSystem`*, and $s \in tss(\mathscr{R}_{\phi,\tau})$ with $s \neq det$, we have $Paths(\mathscr{R}_{\phi,\tau}^{s,nz})_t^{\leq r} \subseteq Paths(\mathscr{R}_{\phi,\tau})_t^{\leq r}$.*

Corollary 1 *For $\mathscr{R}_{\phi,\tau}$, s, r, and t as in Fact 6,*

$$\mathscr{R}_{\phi,\tau}^{s,nz}, L_\Pi, t \not\models_{\leq r} \Phi \quad \textit{implies} \quad \mathscr{R}_{\phi,\tau}, L_\Pi, t \not\models_{\leq r} \Phi.$$

Let $\mathscr{R}_{\phi,\tau}$ be the current module, L_Π a protecting extension of $(\mathscr{R}_{\phi,\tau})^C$ which defines the propositions Π, and let s be the current time sampling strategy. Furthermore, let $L_\Pi^{\hat{C}}$ be the protecting extension of $(\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}}$ which extends L_Π^C by adding the equation

$$\texttt{[}x\ \texttt{in time}\ y\texttt{] |=}\ P = \texttt{true}\ \textbf{if}\ x\ \texttt{in time}\ y\ \texttt{|=}\ P$$

for variables x, y, and P. The time-bounded model checking command

$$\texttt{(mc}\ t_0\ \texttt{|=t}\ \Phi\ \texttt{in time <=}\ r\ \texttt{.)}$$

is interpreted by checking the ordinary LTL satisfaction

$$\mathscr{K}\left((\mathscr{R}_{\phi,\tau}^{s,nz})^{\widehat{\leq r}}, \texttt{[ClockedSystem]}\right)_{L_\Pi^{\hat{C}}}, \texttt{[[}t_0\ \texttt{in time}\ 0_\phi\texttt{]]} \models \Phi$$

using Maude's model checker. The correctness of this choice is given by the following fact:

Fact 7

$$\mathscr{R}_{\phi,\tau}, L_\Pi, t_0 \models_{\leq r} \Phi \quad \textit{if and only if}$$
$$\mathscr{K}\left((\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}}, \texttt{[ClockedSystem]}\right)_{L_\Pi^{\hat{C}}}, \texttt{[[}t_0\ \texttt{in time}\ 0_\phi\texttt{]]} \models \Phi.$$

The validity of this fact is based on the following observations:

- For each path $[t_0\ \texttt{in time}\ r_0] \longrightarrow [t_1\ \texttt{in time}\ r_1] \longrightarrow \cdots$ in $Paths(\mathscr{R}_{\phi,\tau})_{t_0}^{\leq r}$ there is a corresponding path $\texttt{[[}t_0\ \texttt{in time}\ r_0\texttt{]]} \longrightarrow \texttt{[[}t_1\ \texttt{in time}\ r_1\texttt{]]} \longrightarrow \cdots$ in $\mathscr{K}\left((\mathscr{R}_{\phi,\tau})^{\widehat{\leq r}}, \texttt{[ClockedSystem]}\right)_{L_\Pi^{\hat{C}}}$, and vice versa.
- $L_\Pi^C([t\ \texttt{in time}\ r]) = L_\Pi^{\hat{C}}(\texttt{[[}t\ \texttt{in time}\ r\texttt{]]})$ for all terms t and r.

The case where the time bound in a model checking command has the form `<` r is treated in an entirely similar way. The case with bound `no time limit` is model checked by checking whether the L_Π^C-property Φ holds in the rewrite theory $(\mathscr{R}_{\phi,\tau}^{s,nz})^C$.

5.7 Untimed Search and Model Checking

Real-Time Maude also provides commands for *untimed* search and temporal logic model checking, which are particularly useful when the reachable state space from a term $\{t\}$ is finite in $\mathscr{R}_{\phi,\tau}$ but is infinite in $(\mathscr{R}_{\phi,\tau})^C$ due to the time stamps. The untimed commands use the transformation which takes a real-time rewrite theory $\mathscr{R}_{\phi,\tau} = (\Sigma, E, \varphi, R)$ to the rewrite theory $(\mathscr{R}_{\phi,\tau})^U = (\Sigma, E, \varphi, R^U)$, where R^U is the union of the instantaneous rules in R and a rule $l : \{t\} \longrightarrow \{t'\}$ **if** $cond$ for each tick rule of the form $l : \{t\} \xrightarrow{\tau_l} \{t'\}$ **if** $cond$ in R. Since $(\mathscr{R}_{\phi,\tau})^U$ just ignores the durations of tick rules, it follows that the one-step rewrite relations in $(\mathscr{R}_{\phi,\tau})^U$ and in $\mathscr{R}_{\phi,\tau}$ are the same.

Real-Time Maude's untimed search command, with syntax `(utsearch [`n`]` t_0 $arrow$ $pattern$ `.)`, and the untimed model checking command, with syntax `(mc` t_0 `|=u` Φ `.)`, are executed by the corresponding commands in Maude on the rewrite theory $(\mathscr{R}_{\phi,\tau}^{s,nz})^U$ for s the current time sampling strategy. The formula Φ should not contain clocked propositions.

5.8 Other Analysis Commands

The execution of `(find earliest` t_0 `=>*` t `such that` $cond$ `.)` in a module $\mathscr{R}_{\phi,\tau}$, relative to a chosen time sampling strategy s, uses Maude's search capabilities to return a term $\sigma(t)$ `in time` r, such that $\mathscr{R}^{s,nz}_{\phi,\tau} \vdash t_0 \xrightarrow{r} \sigma(t)$ for σ satisfying $cond$, and such that there is no σ' satisfying $cond$ and r' with $r' <_\phi r$ and $\mathscr{R}^{s,nz}_{\phi,\tau} \vdash t_0 \xrightarrow{r'} \sigma'(t)$. The execution of this command may loop if there is no such match σ.

The `(find latest` t_0 `=>*` t `such that` $cond$ $timeBound$ `.)` command (where $timeBound$ is either `with no time limit`, `in time <` r, or `in time <=` r for some time value r) analyzes all behaviors in $\mathscr{R}^{s,nz}_{\phi,\tau}$ and finds the longest time needed, in the worst case, to reach a t-state from t_0. That is, for $timeBound$ of the form `<=` r, the command looks for a $(\mathscr{R}_{\phi,\tau})^C$-term $\sigma(t)$ `in time` r', with σ satisfying $cond$, such that

- for each $\pi \in Paths(\mathscr{R}^{s,nz}_{\phi,\tau})^{\le r}_{t_0}$ there exist σ' (satisfying $cond$), i, and r'' such that $\pi(i)$ equals $[\sigma'(t)$ `in time` $r'']$;
- there exists a (worst) path $\pi \in Paths(\mathscr{R}^{s,nz}_{\phi,\tau})^{\le r}_{t_0}$ and a number i such that $\pi(i)$ equals $[\sigma(t)$ `in time` $r']$ and such that there are no $k < i$, σ' satisfying $cond$, and r'' with $\pi(k) = [\sigma'(t)$ `in time` $r'']$; and
- for each path $\pi \in Paths(\mathscr{R}^{s,nz}_{\phi,\tau})^{\le r}_{t_0}$, if $\pi(i)$ equals $[\sigma'(t)$ `in time` $r'']$ for some i, σ' satisfying $cond$, and r'' with $r'' <_\phi r'$, then there exists a $k < i$ such that $\pi(k) = [\sigma''(t)$ `in time` $r''']$ for some σ'' satisfying $cond$ and r'''.

The cases with $timeBound$ of the forms `<` r and `with no time limit` are defined in a similar way.

For the `check` commands, let p_i be a pattern t_i `such that` $cond_i$, for $i \in \{1,2\}$, where t_i is a ground irreducible term of sort `GlobalSystem` or sort `ClockedSystem`. We can view each p_i as a proposition and can define the labeling function $L_{\{p_1,p_2\}}$ on $(R_{\phi,\tau})^C$-states by $p_i \in L_{\{p_1,p_2\}}([t])$ if and only if there exist a $t' \in [t]$ and a substitution σ satisfying $cond_i$ such that $t' = \sigma(p_i)$. The command `(check` t_0 `|=` p_1 `until` p_2 `in time <=` r `.)` checks the until property

$$\mathscr{R}^{s,nz}_{\phi,\tau}, L_{\{p_1,p_2\}}, t_0 \models_{\le r} p_1 \,\mathtt{U}\, p_2,$$

and the command `(check` t_0 `|=` p_1 `untilStable` p_2 `in time <=` r `.)` checks whether the property p_2 is in addition stable, i.e., it checks the "until/stable" temporal property

$$\mathscr{R}^{s,nz}_{\phi,\tau}, L_{\{p_1,p_2\}}, t_0 \models_{\le r} (p_1 \,\mathtt{U}\, p_2) \mathbin{\texttt{/\textbackslash}} (p_2 \mathtt{=>} \mathtt{[]}\, p_2).$$

The treatment of time bounds of the forms `<` r and `with no time limit` is analogous. Notice that the `find latest` command implicitly contains a check of the liveness property `<>` $pattern$.

The `find latest` and `check` commands are implemented by breadth-first search strategies, and can therefore sometimes decide properties for which the temporal logic model checker fails. In addition, the user does not need to explicitly define temporal logic propositions for these commands. On the minus side, performance may be affected by the fact that these commands do not use Maude's efficient search or model checking facilities.

6 Using Real-Time Maude

In this section we first illustrate specification and analysis in Real-Time Maude with a very simple example (Section 6.1), followed by a more interesting example illustrating object-

oriented specification (Section 6.2) and by a small *hybrid* system example (Section 6.3). Finally, Section 6.4 mentions some larger Real-Time Maude applications.

6.1 A Clock Example

The following timed module models a "clock" which may be running (in which case the system is in state `{clock(`r`)}` for r the time shown by the clock) or which may have stopped (in which case the system is in state `{stopped-clock(`r`)}` for r the clock value when it stopped). When the clock shows `24` it must be reset to `0` immediately:

```
(tmod DENSE-CLOCK is protecting POSRAT-TIME-DOMAIN .
  ops clock stopped-clock : Time -> System [ctor] .
  vars R R' : Time .
  crl [tickWhenRunning] :  {clock(R)} => {clock(R + R')} in time R'
                                        if R' <= 24 - R [nonexec] .
  rl [tickWhenStopped] :
     {stopped-clock(R)} => {stopped-clock(R)} in time R' [nonexec] .
  rl [reset] :   clock(24) => clock(0) .
  rl [batteryDies] :   clock(R) => stopped-clock(R) .
endtm)
```

The two tick rules model the effect of time elapse on a system by increasing the clock value of a running clock according to the time elapsed, and by leaving a stopped clock unchanged. Time may elapse by *any* amount of time less than `24 -` r from a state `{clock(`r`)}`, and by any amount of time from a state `{stopped-clock(`r`)}`. To execute the specification we should first specify a time sampling strategy, for example by giving the command `(set tick def 1 .)`. The command[15]

```
(trew {clock(0)} in time <= 99 .)
```

`Result ClockedSystem : {stopped-clock(24)} in time 99`

then simulates one behavior of the system up to total duration `99`. The command

```
(tsearch [1] {clock(0)} =>* {clock(X:Time)} such that X:Time > 24
             in time <= 99 .)
```

`No solution`

checks whether some state `{clock(`r`)}`, with $r > 24$, can be reached from state `{clock(0)}` in time less than or equal to `99`. Not surprisingly, the *earliest* time the clock can show `10` is after time `10` has elapsed in the system:

```
(find earliest {clock(0)} =>* {clock(10)} .)
```

`Result:    {clock(10)} in time 10`

A corresponding `find latest` search for state `{clock(10)}` will find that there are paths in which the desired state is never encountered:

[15] For each command we also present—in italics—the result of executing the command in Real-Time Maude.

```
(find latest {clock(0)} =>* {clock(10)} in time <= 24 .)

Result: there is a path in which the pattern is not reachable
        in time <= 24
```

Since the reachable state space is finite when we take the time sampling into account, we can check whether a state `{clock(`r`)}`, with r > `24`, can be reached from state `{clock(0)}` by giving the *untimed* search command

```
(utsearch {clock(0)} =>* {clock(X:Time)} such that X:Time > 24 .)

No solution
```

The command

```
(utsearch [1] {clock(0)} =>! G:GlobalSystem .)

No solution
```

shows that there is no deadlock reachable from `{clock(0)}`. Finally, the command

```
(utsearch [1] {clock(0)} =>* {clock(1/2)} .)

No solution
```

will not find the sought-after state, since it is not reachable with the current time sampling strategy.

We are now ready for some temporal logic model checking. The following module defines the *state* propositions `clock-dead` (which holds for all stopped clocks) and `clock-is(`r`)` (which holds if a *running* clock shows r), and the *clocked* proposition `clockEqualsTime` (which holds if the running clock shows the time elapsed in the system):

```
(tmod MODEL-CHECK-DENSE-CLOCK is including TIMED-MODEL-CHECKER .
  protecting DENSE-CLOCK .
  ops clock-dead clockEqualsTime : -> Prop [ctor] .
  op clock-is : Time -> Prop [ctor] .
  vars  R R' : Time .
  eq {stopped-clock(R)}     |=   clock-dead = true .
  eq {clock(R)}             |=   clock-is(R') = (R == R') .
  eq {clock(R)} in time R'  |=   clockEqualsTime = (R == R') .
endtm)
```

The model checking command[16]

```
(mc clock(0) |=u [] ~ clock-is(25) .)

Result Bool : true
```

checks whether the clock is always different from `25` in each computation (relative to the chosen time sampling strategy). The command

[16] Recall that '`|=u`' stands for *untimed* model checking, where the total duration is not taken into account in the analysis.

```
(mc {clock(0)} |=t clockEqualsTime U (clock-is(24) \/ clock-dead)
       in time <= 1000 .)

Result Bool : true
```

checks whether the clock always shows the correct time, when started from `{clock(0)}`, until it shows 24 or is stopped. (Since this latter property involves clocked propositions, we must use the *timed* model checking command.)

Finally, Real-Time Maude's model checker provides a counterexample if the temporal logic property does not hold. For example, it is not always the case that starting from `{clock(0)}` one will always reach a state where the clock shows 3:

```
(mc clock(0) |=u <> clock-is(3) .)

Result ModelCheckResult :
   counterexample({{clock(0)}, 'tickWhenRunning}
                  {{clock(1)}, 'tickWhenRunning}
                  {{clock(2)}, 'batteryDies} ,
                  {{stopped-clock(2)}, 'tickWhenStopped})
```

In this counterexample, the clock ticks (using rule `tickWhenRunning`) to `{clock(2)}`, when the rule `batteryDies` is applied, leading to the state `{stopped-clock(2)}`, from which the system will self-loop forever using rule `tickWhenStopped`.

6.2 An Object-Based Network Protocol Example

We illustrate real-time object-oriented specification with a protocol for computing *round trip times* (i.e., the time it takes for a message to travel from an initiator node to a responder node, and back) between pairs of nodes in a network. The setting is simplified to illustrate key features of object-oriented real-time specifications—such as timers and the functions `delta` and `mte`—without drowning the reader in details. A Real-Time Maude specification of a "real" protocol for estimating round trip times is given as part of the specification of the AER/NCA protocol suite [29].

The setting is simple: each node is interested in finding the round trip time to exactly one other node. Communication is modeled very generally by "ordinary" message passing, where it may take a message *any* amount of time to travel from one node to another.

The protocol is equally simple: An initiator object o has a local clock and starts a run of the protocol by sending an `rttReq` message to its neighbor o' with its current time stamp r (rule `startSession`). When the neighbor o' receives the `rttReq` message, it replies with an `rttResp` message, to which it attaches the received time stamp r (rule `rttResponse`). When the initiator node o reads the `rttResp` with its original time stamp r, the rtt value is just its current clock value minus the original time stamp r (rule `treatRttResp`).

One problem with this version of the protocol is that it may happen that the response message is not received within reasonable time. In such cases it is appropriate to assume that there is a problem with the message delivery. Therefore, only round trip times less than a time value `MAX-DELAY` are considered (rule `ignoreOldResp` ignores responses which are too old). If the initiator does not receive a response in time less than `MAX-DELAY`, it has to initiate another round of the protocol exactly time `MAX-DELAY` after its first attempt (rule `tryAgain`).

The process is repeated until an rtt value less than `MAX-DELAY` is found. A `findRtt(`o`)` message “kicks off” a run of the protocol for object o.

In the following specification, each `Node` object uses a `timer` attribute to ensure that a new attempt is initiated at every `MAX-DELAY` time units, until an rtt value is found. If the timer has value r, it must “ring” in time r from the current time. The timer is turned off when its value is `INF`. The class `Node` has the attributes `nbr`, which denotes the node whose rtt value it is interested in, and a `clock` attribute denoting the value of its local clock. The `rtt` attribute stores the rtt to its preferred neighbor:

```
(tomod RTT is protecting NAT-TIME-DOMAIN-WITH-INF .
  op MAX-DELAY : -> Time .   eq MAX-DELAY = 4 .

  class Node | clock : Time, rtt : TimeInf,
               nbr : Oid, timer : TimeInf .

  msgs rttReq rttResp : Oid Oid Time -> Msg .
  msg  findRtt : Oid -> Msg .                --- start a run

  vars O O' : Oid .   vars R R' : Time .   var TI : TimeInf .

  --- start a session, and set timer:
  rl [startSession] :
     findRtt(O)  < O : Node | clock : R, nbr : O' >  =>
                 < O : Node | timer : MAX-DELAY >   rttReq(O', O, R) .

  --- respond to request:
  rl [rttResponse] :
     rttReq(O, O', R)  < O : Node | >  =>
                       < O : Node | >  rttResp(O', O, R) .

  --- received resp within time MAX-DELAY;
  ---   record rtt value and turn off timer:
  crl [treatRttResp] :
      rttResp(O, O', R)  < O : Node | clock : R' >  =>
                         < O : Node | rtt : (R' monus R), timer : INF >
      if (R' monus R) < MAX-DELAY .

  --- ignore and discard too old message:
  crl [ignoreOldResp] :
      rttResp(O, O', R)  < O : Node | clock : R' >  =>  < O : Node | >
      if (R' monus R) >= MAX-DELAY .

  --- start new round and reset timer when timer expires:
  rl [tryAgain] :
     < O : Node | timer : 0, clock : R, nbr : O' >  =>
     < O : Node | timer : MAX-DELAY >  rttReq(O', O, R) .

  --- tick rule should not advance time beyond expiration of a timer:
  crl [tick] :
      {C:Configuration}  =>  {delta(C:Configuration, R)} in time R
      if R <= mte(C:Configuration) [nonexec] .

  --- the functions mte and delta:
  op delta : Configuration Time -> Configuration [frozen (1)] .
  eq delta(none, R) = none .
  eq delta(NEC:NEConfiguration NEC':NEConfiguration, R) =
       delta(NEC:NEConfiguration, R) delta(NEC':NEConfiguration, R) .
  eq delta(< O : Node | clock : R, timer : TI >, R') =
           < O : Node | clock : R + R', timer : TI monus R' > .
```

```
  eq delta(M:Msg, R) = M:Msg .

  op mte : Configuration -> TimeInf [frozen (1)] .
  eq mte(none) = INF .
  eq mte(NEC:NEConfiguration NEC':NEConfiguration) =
        min(mte(NEC:NEConfiguration), mte(NEC':NEConfiguration)) .
  eq mte(< O : Node | timer : TI >) = TI .
  eq mte(M:Msg) = INF .
endtom)
```

This use of timers, clocks, and the functions `mte` and `delta` is fairly typical for object-oriented real-time specifications. Notice that the tick rule may advance time when the configuration contains messages. The following timed module defines an initial state with three nodes `n1`, `n2`, and `n3`:

```
(tomod RTT-I is including RTT .
  ops n1 n2 n3 : -> Oid .
  op initState : -> GlobalSystem .
  eq initState =
       {findRtt(n1) findRtt(n2) findRtt(n3)
        < n1 : Node | clock : 0, timer : INF, nbr : n2, rtt : INF >
        < n2 : Node | clock : 0, timer : INF, nbr : n3, rtt : INF >
        < n3 : Node | clock : 0, timer : INF, nbr : n1, rtt : INF >} .
endtom)
```

The reachable state space from `initState` is infinite, since the time stamps and clock values may grow beyond any bound and the state may contain any number of old messages. Search and model checking should be time-bounded to ensure termination. We set the time sampling strategy with the command `(set tick def 1 .)` to cover the discrete time domain.

The command

```
  (tsearch [1]
      initState  =>* {C:Configuration
                      < O:Oid : Node | rtt : X:Time,
                                        ATTS:AttributeSet >}
         such that X:Time >= 4
            in time <= 10 .)

  No solution
```

checks whether a state with an undesired `rtt` value $\geq$ `4` can be reached within time `10`. The command

```
  (tsearch [1]
      initState =>* {C:Configuration
                     < n1 : Node | rtt : 2, ATTS:AttributeSet >
                     < n2 : Node | rtt : 3, ATTS':AttributeSet >}
         in time <= 5 .)

  Solution 1
    ATTS':AttributeSet <- clock : 3, nbr : n3, timer : INF ;
    ATTS:AttributeSet <- clock : 3, nbr : n2, timer : INF ;
    C:Configuration <-
       findRtt(n3)
       < n3 : Node | clock : 3, nbr : n1, rtt : INF, timer : INF > ;
    TIME_ELAPSED:Time <- 3
```

checks whether a state with `rtt` values `2` and `3` can be reached.

We illustrate temporal logic model checking by proving that there are no *superfluous* messages being sent around in the system after an `rtt` value has been found. That is, if an object o has found an `rtt` value, then there is no `rttReq(`o', o, r`)` or `rttResp(`o, o',r`)` message with $r+$`MAX-DELAY` $> c$, for c the value of o's `clock`. The following module defines the proposition `superfluousMsg`:

```
(tomod MC-RTT is including TIMED-MODEL-CHECKER .  protecting RTT-I .
  op superfluousMsg : -> Prop [ctor] .
  vars REST : Configuration .   vars O O' : Oid .   vars R R' R'' : Time .

  ceq {REST < O : Node | rtt : R, clock : R' >  rttReq(O', O, R'')}
          |= superfluousMsg  =  true     if R'' + MAX-DELAY > R' .
  ceq {REST < O : Node | rtt : R, clock : R' >  rttResp(O, O', R'')}
          |= superfluousMsg  =  true     if R'' + MAX-DELAY > R' .
endtom)
```

The command

```
  (mc initState |=t [] ~ superfluousMsg in time <= 10 .)

  Result Bool : true
```

proves that there are no superfluous messages in the system within time `10`. More interesting temporal properties about similar specifications are given in [24]; examples of sophisticated Real-Time Maude model checking are provided in [29].

6.2.1 Modeling Different Message Transmission Delays

In the above model, the transmission of a message can take *any* amount of time $\geq$ `0`. The equation

```
  eq mte(M:Msg) = INF .
```

implies that time progress is not impeded by the presence of messages in the configuration, thus allowing a message to remain "forever" in the configuration without being read. As for the lower bound, we see that, e.g., an `rttReq` message created in the rules `startSession` and `tryAgain` can be read in the rule `rttResponse` without the tick rule having been applied in-between.

In this section we show how to modify the module `RTT` to model the settings where:

1. it takes a message *at least* time `MIN-TRANS-TIME` to travel from its source to its destination; and
2. it takes a message *exactly* time `MIN-TRANS-TIME` to travel from source to destination.

In addition, we will briefly indicate how to model message transmission times in more detail by considering the physical properties of the *links* through which the messages travel.

To model "delay" in message transmission, we add a delay operator `dly` of a supersort `DlyMsg`. The meaning of `dly(`m`,`r`)` is that the message m will be "ripe" in time r. That is, it will become m in time r. It is obvious that we want `dly(`m`,0) =` m, so the delay operator is declared to have *right identity* `0`:

```
  sort DlyMsg .      subsorts Msg < DlyMsg < NEConfiguration .
  op dly : Msg Time -> DlyMsg [ctor right id: 0] .
```

To send a message m which will take at least time `MIN-TRANS-TIME` to reach its destination, the message `dly(`m`, MIN-TRANS-TIME)` should be sent. For example, the *right-hand* side of the rules `tryAgain` and `startSession` should in this case be

```
< O : Node | timer : MAX-DELAY >
dly(rttReq(O', O, R), MIN-TRANS-TIME) .
```

The *left-hand* sides of the message-consuming rules should not change: only ripe messages should be read. The equation defining the function `delta` on single messages must be replaced by the equation

```
eq delta(dly(M:Msg, R), R') = dly(M:Msg, R monus R') .
```

(This equation also applies to ripe messages, since m = `dly(`m`, 0)` follows from `dly` being declared to have right identity `0`.) This technique models *minimum* transmission delay in message passing communication.

To model setting (i), where the *maximum* possible message transmission is unbounded, we use the equation

```
eq mte(DM:DlyMsg) = INF .
```

For setting (ii), where the *exact* message transmission time equals the smallest possible transmission time, we replace the above equation for `mte` by

```
eq mte(dly(M:Msg, R)) = R .
```

so that the `mte` of a ripe message is `0` (again, due to the right identity of `dly`). With the last equation, time cannot advance when a ripe message is present in the configuration, forcing ripe messages to be treated without delay.

The manual [24] presents these versions—as well as more sophisticated ones—of our RTT example in detail.

Links. An alternative way of modeling communication is to use explicit *link objects*, inside which packets travel from source to destination. Such a more detailed model of links—where the delay of a packet is given as a function of the propagation delay and the speed of the link, the delays of the other packets in the link, and the size of the packet—was needed in the AER/NCA case study, and is described in [29, Section 4.6.1].

6.3 A Hybrid System Example: A Thermostat

We finish our collection of examples with a small *hybrid* system example: A thermostat works by turning on and off a heater in order to maintain a temperature between 62 and 74 degrees. When the heater is turned off, the temperature decreases by *one degree* per time unit, and when the heater is turned on the temperature increases by *two degrees* per time unit.[17] In addition, the thermostat is equipped with a "stopwatch" which keeps track of the total time that the heater has been turned *on*, so that the local energy company can charge the correct amount to the user.

Assuming that the time and temperature domains can be modeled by the nonnegative rational numbers, a Real-Time Maude specification of the thermostat can be given as follows, where $l\,,x\,,d$ denotes the state of the system, with x the current temperature, l the current control state (either `on` or `off`), and d the total duration that the heater has been on.

[17] For simplicity, we use linear functions to describe temperature increases or decreases. More complex dynamics can also be modeled in Real-Time Maude by defining the necessary functions.

```
(tmod THERMOSTAT is
  protecting POSRAT-TIME-DOMAIN .   --- Dense time domain
  sort ThermoState .
  ops on off : -> ThermoState [ctor] .

  op _`,_`,_ : ThermoState PosRat PosRat -> System [ctor] .

  vars R R' R'' : Time .

  rl [turn-on] :   off, 62, R  =>  on, 62, R .
  rl [turn-off] :  on, 74, R  =>  off, 74, R .

  crl [tick-on] :
      {on, R, R'}  =>  {on, R + (2 * R''), R' + R''} in time R''
      if R'' <= ((74 - R) / 2) [nonexec] .

  crl [tick-off] :
      {off, R, R'}  =>  {off, R - R'', R'} in time R''
      if R'' <= (R - 62) [nonexec] .
endtm)
```

This system, with its uninitialized “stopwatch,” cannot be expressed by timed automata or by decidable classes of hybrid automata [16].

6.4 Some Real-Time Maude Applications

Real-Time Maude is particularly suitable for specifying distributed systems in an object-oriented style. All our larger Real-Time Maude applications have, as mentioned above, been so specified. They include the formal specification and analysis of:

- The new and sophisticated AER/NCA suite of protocols [18] that intend to achieve reliable, scalable, and TCP-friendly multicast in active networks. Real-Time Maude analysis uncovered subtle design errors which could not be found by traditional testing by the protocol developers, while independently finding all bugs discovered by such testing [29].
- The NORM multicast protocol developed by the Internet Engineering Task Force [20].
- A series of new scheduling algorithms, with advanced capacity sharing facilities, for real-time systems [25].
- Advanced wireless sensor network protocols [30].

In addition, we showed in [28] that real-time rewrite theories can be seen as a semantic framework in which a wide range of models of real-time and hybrid systems can be naturally represented. Therefore, Real-Time Maude has the potential to serve as an execution and analysis environment for other real-time formalisms not having tools of their own. Thus far, an execution environment for a real-time extension of the Actor model has been developed [13].

7 Concluding Remarks

We have presented Real-Time Maude, have described and illustrated its features, and have documented the tool's semantic foundations. Perhaps the most important lesson learned is that formal specification and analysis of real-time systems—including distributed object-based systems with real-time features—can be supported with good expressiveness and with reasonable efficiency in important application areas outside the scope of current decision procedures. What seems desirable for system design purposes is to have a *spectrum* of analysis methods that spans automated verification on one side and simulation and testbeds on the other. We view Real-Time Maude as addressing the middle area of this spectrum, and providing a good semantic basis for integrating other methods on the spectrum's edges in the future.

Several research directions should be investigated in the near future:

1. the current incomplete analyses due to choices in the time sampling strategies should be made complete by identifying useful system classes for which such strategies are complete, and by developing new abstraction techniques;
2. the use of Real-Time Maude specifications to generate code meeting desired real-time requirements should be investigated; and
3. symbolic reasoning and deductive techniques complementing the current analysis capabilities should be developed.

Of course, all these future developments should be driven by new applications and case studies. We hope that the current tool will stimulate users to contribute their ideas and experience in advancing the research areas mentioned above and many others.

Acknowledgments: We are grateful to Narciso Martí-Oliet, Miguel Palomino, Carolyn Talcott, Alberto Verdejo, and the anonymous referees for many helpful comments on earlier versions of this paper. Partial support of this research by ONR Grant N00014-02-1-0715, by NSF Grant CCR-0234524, and by The Norwegian Research Council is gratefully acknowledged.

References

1. W. M. P. van der Aalst. Interval timed coloured Petri nets and their analysis. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 453–472. Springer, 1993.
2. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
3. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
4. R. Alur and T.A. Henzinger. Logics and models of real time: A survey. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer, 1992.
5. G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Proc. Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004. See also UPPAAL home page at `http://www.uppaal.com`.
6. M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. Tools and applications II: The IF toolset. In M. Bernardo and F. Corradini, editors, *Proc. Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, volume 3185 of *Lecture Notes in Computer Science*, pages 237–267. Springer, 2004.

7. R. Bruni and J. Meseguer. Generalized rewrite theories. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2003.
8. E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
10. M. Clavel, F. Dúran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.1.1)*, April 2005. `http://maude.cs.uiuc.edu`.
11. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Reflection'96*, pages 263–288, 1996. `http://jerry.cs.uiuc.edu/reflection/`.
12. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.
13. H. Ding, C. Zheng, G. Agha, and L. Sha. Automated verification of the dependability of object-oriented real-time systems. In *Proc. 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03)*. IEEE Computer Society Press, 2003.
14. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Fourth International Workshop on Rewriting Logic and its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.
15. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
16. T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.
17. G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
18. S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. *IEEE Network Magazine (Special Issue on Multicast)*, 14(1):48–57, 2000.
19. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
20. E. Lien. Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude. Master's thesis, Department of Linguistics, University of Oslo, 2004.
21. Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 30:609–678, 1993.
22. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.
23. P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, 2000. Available at `http://maude.cs.uiuc.edu/papers`.
24. P. C. Ölveczky. *Real-Time Maude 2.1 Manual*, 2004. `http://www.ifi.uio.no/RealTimeMaude/`.
25. P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering (FASE'06)*, volume 3922 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2006.
26. P. C. Ölveczky, M. Keaton, J. Meseguer, C. Talcott, and S. Zabele. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE 2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2001.
27. P. C. Ölveczky and J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In K. Futatsugi, editor, *Third International Workshop on Rewriting Logic and its Applications*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.
28. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.
29. P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 2006. To appear.
30. P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*. IEEE Computer Society Press, 2006.
31. P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.
32. S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.